Models and resolution principles for logical meta-programming languages

H. Christiansen

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

1 9 9 2

25 *ème anniversaire*

N° 1594

Programme 2
Calcul Symbolique, Programmation
et Génie logiciel

MODELS AND RESOLUTION PRINCIPLES FOR LOGICAL META-PROGRAMMING LANGUAGES

Henning CHRISTIANSEN

Février 1992

RR.1594

Models and resolution principles for logical meta-programming languages

Henning Christiansen

Abstract.
Meta-programming extends logic programming with the possibility of having a program to create or analyze other programs represented as data structures and to give queries dynamically to these programs. Thus languages for meta-programming represent a logical formalism in which logical theories can refer to provability in arbitrary other, perhaps dynamically generated, theories.

The importance is stressed of allowing variables in the program representation which stand for unknown program fragments, even in the program currently under execution. This can be used, for example, for hypothetical reasoning where a program generates the hypotheses to be tested or for program generation in the sense that an interpreter fills in the missing program parts. It may lead to a more declarative style of, e.g., program generation — the meta-program can specify properties of the desired programs at a higher level of abstraction and the detailed synthesis is put in the hands of an implementation.

The declarative semantics is described in terms of Herbrand-models analogous to those normally used for plain logic programs and the well-known fixed-point results generalize immediately. The procedural semantics is given as a resolution method which is shown to be sound and complete. Complete resolution for meta-programming languages is more complicated than usual due to the possibility of uninstantiated variables which stand for unknown parts of the program currently under execution.

Modèles et principes de résolution pour les langages de méta-programmation en logique

Henning Christiansen

Résumé
La méta-programmation étend la programmation en logique en permettant à un programme de créer ou d'analyser d'autres programmes représentés comme des structures de données et d'adresser dynamiquement des requêtes à ces programmes. Ainsi les langages de méta-programmation constituent un formalisme logique dans lequel des théories logiques peuvent se référer à la prouvabilité dans n'importe quelles autres théories, éventuellement créées dynamiquement.

Il est particulièrement important de permettre l'utilisation de variables dans la représentation des programmes pour dénoter des fragments de programme inconnus, et ceci même dans le programme en cours d'exécution. Cela peut par exemple servir pour le raisonnement hypothétique où un programme engendre les hypothèses à tester ou pour la génération de programmes dans la mesure où un interprète remplit les portions manquantes du programme. Cela peut conduire à un style plus déclaratif, par exemple pour la génération de programmes: le méta-programme peut spécifier les propriétés des programmes désirés à un plus haut niveau d'abstraction et la synthèse précise est l'affaire d'une implémentation.

La sémantique déclarative est décrite en termes de modèles de Herbrand, analogues à ceux habituellement utilisés pour les programmes logiques et les résultats de point fixe bien connus se généralisent immédiatement. La sémantique opérationnelle est donnée sous la forme d'une méthode de résolution dont on démontre la correction et la complétude. Une résolution complète pour les langages de méta-programmation est plus compliquée que la résolution usuelle du fait de la possibilité d'avoir des variables non instanciées représentant des portions inconnues du programme en cours d'exécution.

Models and resolution principles for logical meta-programming languages

Henning Christiansen
Roskilde University
P.O. Box 260, DK-4000 Roskilde, Denmark
E-mail: henning@dat.ruc.dk

Abstract. Meta-programming extends logic programming with the possibility of having a program to create or analyze other programs represented as data structures and to give queries dynamically to these programs. Thus languages for meta-programming represent a logical formalism in which logical theories can refer to provability in arbitrary other, perhaps dynamically generated, theories.

The importance is stressed of allowing variables in the program representation which stand for unknown program fragments, even in the program currently under execution. This can be used, for example, for hypothetical reasoning where a program generates the hypotheses to be tested or for program generation in the sense that an interpreter fills in the missing program parts. It may lead to a more declarative style of, e.g., program generation — the meta-program can specify properties of the desired programs at a higher level of abstraction and the detailed synthesis is put in the hands of an implementation.

The declarative semantics is described in terms of Herbrand-models analogous to those normally used for plain logic programs and the well-known fixed-point results generalize immediately. The procedural semantics is given as a resolution method which is shown to be sound and complete. Resolution for meta-programming languages is more complicated than usual due to the possibility of uninstantiated variables which stand for unknown parts of the program currently under execution.

1 Introduction

In this paper, we study the semantics of logical meta-programming languages. The distinguishing feature of a meta-programming language is the representation of the language's syntax and semantics within itself. So, for example, a program may analyze or synthesize other programs or perhaps manage sets of other programs. For applications of meta-programming techniques see, e.g., (Kowalski, 1979, Bowen, Kowalski, 1982, Bowen, 1985, Abramson, Rogers, 1989, Bruynooghe, 1990).

We describe the declarative and procedural semantics for such languages in which we allow logical variables as placeholders for program fragments. This means that we can describe hypothetical reasoning in which a program generates the hypotheses to be tested. Or an interpreter can be used as a program generator, queries can be executed in, perhaps partially or completely, unknown programs and the implementation completes the program as much as needed.

More precisely, we study the semantics of logical meta-programming languages with reflexive constructs corresponding the classical, binary demo-predicate,

demo(*goal*, *program-representation*).

It takes as arguments a goal and a ground representation of a program and it succeeds if the goal can be proved in the program represented by the second argument. A variable here

will thus represent an unknown piece of program text. In the literature about these matters, however, there seems to be a tacit agreement that the second argument always is instantiated to a ground value, i.e., it represents a determinate program, when a call of demo is reached by execution. In the paper by Bowen and Kowalski (1982) which introduced the demo predicate, the semantics is specified by a sketch of an interpreter. Unfortunately, the delicate parts of the interpreter that might give a hint on how to treat such variables are omitted. In Hill and Lloyd's (1989) more detailed study of such interpreters, alas, the program clauses are, as a formal manipulation, represented by global facts. Thus, they can only be completely given. Assumptions of this sort, that some variables must be grounded when certain goals are called, are usually not appreciated in the influential circles of logic programming for several good reasons.

- The intended, procedural semantics is obviously not complete.
- A very operational view, considering the variables' order of instantiation, is needed in order to understand a program.

This state-of-the-art is not only unsatisfactory from a formalistic point of view, it also precludes a vast potentiality of interesting computations. Why should an interpreter only be used for executing programs? A complete implementation may also create new programs, or finish program sketches — by providing suitable values for variables which stand for program fragments. We may thus hope for a more declarative style of meta-programming in the sense that our meta-programs can change from the present-day detailed synthesis/analysis into specifying more abstract properties about programs. In this paper we provide at least a basis giving a declarative as well as a sound and complete, procedural semantics.

Our approach is described for language called generative clause programs, or GCP, introduced in (Christiansen, 1990a). It corresponds to definite clause programs, i.e., plain, pure Prolog programs, extended with a demo(–, –) predicate, but it has a more uniform structure which simplifies the formal expositions. Consider, as an example, the following definite clause whose body contains a call of a demo predicate.

p :– q, demo(r, *Prog*)

It specifies that goal p can be proved if q and r can be proved; q in the same program as p, r in the program represented by the term *Prog*. In our language, this clause is written as follows.

p in Var :– q in Var, r in *Prog*.

When this rule is chosen for the execution of p, the variable Var will be bound to a representation of the current program which, then, is carried over to q — whereas r is executed in another program, *Prog*. The syntactic constellation of goal and program representation is called a meta-goal and it serves as the fundamental syntactic and semantic category in our approach.

We use a ground representation of programs within themselves, i.e., in the sample clause above, the variables in *Prog* are represented by special constant symbols. An actual variable in *Prog*, on the other hand, would represent an unknown piece of program text. Consider, as an example, the following meta-goal.

p(X) in [··· p(*x) :– Y ···]

The X in the goal p(X) is a variable which communicates values between this meta-goal and its context, the *x is a constant which denotes a variable, which, then, serves as a locally quantified variable, whereas Y is a variable which communicates values that represent program fragments. As we will argue in section 2, the ground representation implies a dynamic potentiality for program synthesis which is not present if implications as goals and explicit

quantification is used instead; Giordano, Martelli, and Rossi (1991) provides a study of the semantics of this sort of languages.

Now, this uniform, syntactic structure makes it quite straightforward to generalize the model-based, declarative semantics of DCP (or pure Prolog). We propose Herbrand models consisting of ground meta-goals and the usual fixed-point characterization of the least such model generalizes immediately.

In a similar way, we generalize the well-known SLD-resolution principle for DCP to GCP. For reasons of analogy, we talk about SLG-resolution. It consists roughly of the same steps as ordinary SLD-resolution for plain logic programs (van Emden, Kowalski, 1976, Lloyd, 1987). However, new techniques are needed in order to handle selection of clauses and goals from not yet known programs and to represent the renaming of variables etc. for such objects. Our resolution states are systems of equations between suitably generalized terms. Unification is made by an adaptation of the algorithm given by Martelli and Montanari (1982). ***However, the termination of our unification algorithm has not been proved yet and remains, thus, as a conjecture.***

Finally, we present a limited version of the resolution method, which we call weak SLG-resolution, for the special cases in which no non-ground parts appears. Here our resolution method collapses to a simple interpreter in the style described by Bowen and Kowalski (1982) and studied in more detail by Hill and Lloyd (1989). This is obviously not complete but it anticipates efficient implementations, e.g., in the style of Bacha (1987, 1988). This is basically compilation *á la* Prolog extended with separate (but possibly structure sharing, Christiansen, 1989b) program spaces and with dynamic calls of the compiler when needed.

Overview

In section 2, we give an overview of other work concerned with the semantics of logical meta-programming languages and related topics. Section 3 explains our choice of a syntax for a meta-programming language. Section 4 gives the model-based semantics. Due to the uniform syntactic structure of GCP, our developments become a mere rephrasing of the standard results for DCP (Lloyd, 1987).

Section 5 on SLG-resolution is far the most difficult part of the paper since it have to develop a theory of suitably generalized terms, substitutions and unification from scratch in order to cope with the complexity imposed by the execution of unknown program fragments.

The final section provides for a summary and some discussion together with a longer example of a resolution process — which we also use to illustrate the need for the development of a methodology for using the resolution method.

Acknowledgments

Parts of this work has been made at the author's visit at INRIA, Rocquencourt, France, partially supported by the Eureka Software Factory Project (ESF) and by the Danish Natural Science Research Council.

The model-based, declarative semantics and the fixed-point results have been presented at the Second Workshop on Meta-programming in Logic in Leuven, Belgium, (Christiansen, 1990a).

2 Related work

Levels of expressibility

Provability and truth predicates has been studied extensively within mathematical logic, especially in modal logic. These result, however, cannot be directly applied to the general meta-

programming situation. The "classical" results — (Kripke, 1975) is a primary reference — are concerned with truth statements about a *given* theory (or program), and truth statements about such truths, etc. Reflexive meta-programming in the sense considered in the present paper is concerned with systems of many concurrent and equally important theories — and each theory may access other theories in arbitrary ways; see fig. 1 below.

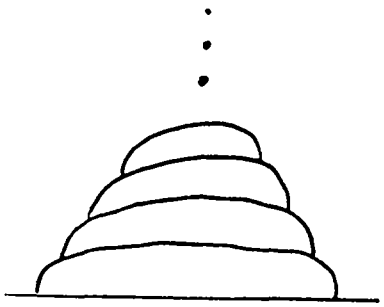

Fig. 1 a. Classical truth predicates

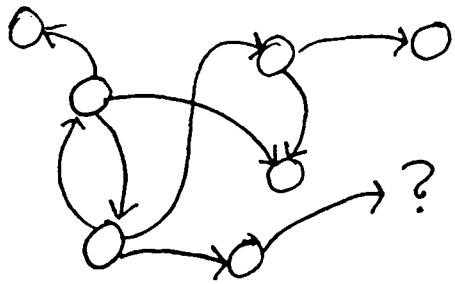

Fig. 1 b. General meta-programming

Kripke (1975) has described Herbrand-like models for logical languages equipped with such a truth predicate. The same situation has been studied in modal logic as necessity (the box operator) can be interpreted as provability (e.g., Boolos, 1979, Smoryński, 1984, 1985).

An obvious extension of first order logic towards a more general notion of provability is to use the well-known deduction theorem (e.g., Shoenfield, 1967, or Enderton, 1972) which can be phrased as follows.

A is provable in theory T extended with F iff $F \Rightarrow A$ is provable in T.

This makes possible a simple form of hypothetical reasoning: "If it were the case that F (in addition to what we already know), what about A?". In general, this principle provides a method for *local extensions* of the current theory. We can illustrate this by another intuitive diagram, fig. 2, indicating an expressibility in between fig. 1 a and 1 b.

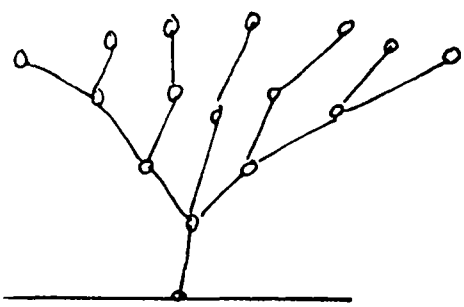

Fig. 2. Local extensions of current theory

Logic programming languages with facilities for extending the current program with particular formulas has been studied, among others, by (Gabbay, Reyle, 1984), (Warren, 1984), (Nait Abdallah, 1986, 1989), (Miller, 1986, 1989), (Monteiro, Porto, 1989), and (Giordano, Martelli, Rossi, 1991). Referring to the deduction theorem, this may be shaped linguistically by allowing implications as goals in the body of clauses, e.g.,

p :– ..., $F \Rightarrow A$, ...

These proposed extensions are, however, purely static, the programs as well as their extensions has to be given in the program text.

From a strictly logical point of view, it may be cleaner to consider the language as consisting of (particular) formulas of first order logic with provability encoded as implication and the scope of each variable indicated by explicit quantification. On the other hand, this excludes the possibility to have the clauses to be added (i.e., the "F" above) synthesized in an arbitrary fashion by other predicates. A variable, X, is only meaningful within the scope given by a quantification of X — it cannot be treated as plain data when putting a new clause together. Using a ground representation as proposed in Gödel (Burt, Hill, Lloyd, 1990) or the present paper, a *constant* such as *x (which denotes a variable) can appear everywhere and the full power of logic programming can be used to synthesize new program clauses. So languages with implication goals are more like logic programming with modules.

The programs in such languages can be seen as special cases of generative clause programs, in which each rule takes the form as follows; the ampersand represents program extension.

$$\cdots \text{in } \mathit{Context} \text{ :- } \cdots \text{in } \mathit{Context} \,\&\, \mathit{static\text{-}extension}, \cdots$$

Correspondingly, semantic definitions (e.g., Nait Abdallah, 1986, 1989, Miller, 1986, Giordano, Martelli, Rossi, 1991) for these languages can be seen as special cases of our Herbrand-models.

Our semantic models are related to Nait Abdallah's (1986, 1989) models for a logical language with modules (called "procedures"). His models consist of pairs, called ions, of goal and program parts and he obtains a fixed-point characterization of a least Herbrand-model similar to ours. Broghi, Lamma, and Mello (1990) consider the semantics of a language identical to Nait Abdallah's and give an almost identical, model-based semantics; the only difference is that the program components of their semantic objects are sets of sub-program names instead of the clauses themselves. They also give a resolution method which is related to our weak SLG-resolution. Goguen and Meseguer (1984) describe an extension of DCP with equality and generic modules. The paper does not give an explicit semantic definition for the module concepts, but refers to it as a special case of "institutions" (Goguen, Burstall, 1984).

Approaches related to the semantics of meta-programming languages

For languages with implication goals (i.e., static modules, cf. above), Giordano, Martelli, and Rossi (1991) have shown how different interpretations of the implication symbol lead to languages with either dynamic or static visibility rules or with "closed modules" which corresponds to a static version of the demo(–, –) predicate. They show how possible worlds — or Kripke — models (Kripke, 1963, Boolos, 1979, Smoryński, 1973) can be given for each of these languages. In (Giordano, Martelli, 1991) it is furthermore shown that each of these languages can be interpreted within the modal logic S4 (see, e.g., Boolos, 1979), letting the operators in the logical languages stand for different combinations of modal operators.

The only known approach which addresses the semantics of logical meta-programming languages with built-in reflexive constructs of the sort we have in mind is made by Subramanian (1989). He considers a language with a *unary* demo predicate in the style of (Kripke, 1975). Provability of a goal, g, in an *arbitrary* theory T, i.e., binary demo, is with reference to the deduction theorem attempted described follows.

$$\mathrm{demo}(\ulcorner g \Leftarrow T \urcorner)$$

(The brackets denote Gödel numbering, and the theory is identified with the conjunction of its clauses). The unary demo satisfies the following axioms.

$$\mathrm{demo}(\ulcorner V \urcorner) \Leftarrow V$$

$$\mathrm{demo}(\ulcorner V \urcorner) \Leftarrow \mathrm{demo}(\ulcorner V \Leftarrow U \urcorner) \wedge \mathrm{demo}(\ulcorner U \urcorner)$$

Due to the first axiom, the binary demo amounts, not to provability in T, but in the "current" theory extended with T. I.e., the approach is concerned with static extension (fig. 2) and not with provability in arbitrary theory (fig. 1 b). A model-based semantics is given which is not based on Herbrand models; for some reason, model forcing (Robinson, 1971, Marek, 1988) is used in the referenced approach in order to ensure the desired continuity and fixed-point properties.

The straightforward use of Gödel numbering, although well-suited for theoretical studies, should also be avoided in a programming language since the "Gödel brackets" serve as a quotation mechanism and there is no corresponding "un-quote" to take care of the communication of values, whether this be in the goal part or variables which stand for program fragments.

There are several examples of model-based semantic definitions for languages which extend DCP with constructs for accessing the individual components of a goal, e.g., the predicate symbol, thereby enabling a sort of higher order logic. See, e.g., (Costantini, 1990, Sugano, 1990). The usual fixed-point semantics generalizes immediately in these cases. However, this comes as no surprise since this expressibility can be simulated in DCP writing goals as lists, i.e., write "[p, x, y, z]" instead of "p(x, y, z)". In this way it is possible to write "meta-level" or "reflective" clauses of the following sort.

[Pred | Args] :- ...

Therefore we have not paid attention to such facilities in our choice of a syntax and in our semantic models.†

However, we may object that these languages support only the syntactic aspect of meta-programming giving tools for elaborating the syntactic structures of the language. The semantic part, on the other hand, i.e., the execution of generated program text is not covered, although the syntactic tools are present which in principle make it possible to write an interpreter. Our own experience with the development of a resolution method of full generality shows that writing such an interpreter will be a very complex task.

The Reflective Prolog language (Costantini, Lanzarone, 1988, Costantini, 1990) provides an interesting possibility at the semantic level. Here new clauses can *extend* the underlying interpreter to take special care of, say, predicates which are symmetric.

The Gödel language (Burt, Hill, Lloyd, 1990) supports the syntactic portion of meta-programming by means of an elaborate type system together with a catalogue of auxiliary predicates for meta-programming. The semantic representation is not supported in Gödel.

An often used method to describe the meaning of programs which call a demo predicate is to program the demo predicate in the logic language at hand, i.e., to write an interpreter for it. This may be an acceptable way to specify a *predicate* with the arbitrary name demo, but it is not a good way to define the semantics of a meta-programming *language*. It is not satisfactory from either a theoretical or practical point of view, since the models (and proofs!) thus become dominated by statements about the interpreters internal mode of operation — blurring what actually is being proved. Furthermore, none of the interpreters presented in the literature describe how to handle uninstantiated variables which stand in the position of a program fragments. In Bowen and Kowalski's (1982) interpreter for the binary demo predicate it is unclear what the meaning of a call with a non-ground program argument should be since the more delicate parts of the interpreter are left unspecified. In Hill and Lloyd's (1989) more detailed studies of interpreter algorithms, the program argument is implicitly assumed to be

† This is, on the other hand, not an argument for not having clean facilities in a practical programming language for this purpose! See, e.g., (Costantini, Lanzarone, 1988).

ground since it is not actually given as an argument but stored in global facts which can be consulted by the interpreter. Moreover, it is difficult to give a satisfactory treatment of the case where the interpreter is called from interpreted code.

These problems do not arise in our declarative semantic approach since each goal carries the program in which it is being proved — and no interpreter program is mixed up with the semantics of the language. The confusing distinction between "object-level" and "meta-level" computation made in the works referenced above disappears.

In the choice of the generative clause language, we have concentrated on having an as simple and uniform structure as possible, in order to achieve a clear semantics. The syntactic denotation function which maps ground program representations to programs (with variables) eliminates the need for a type system or special predicates. Again, this simplifies the semantic definitions but the robustness of a type system such as Gödel's (Burt, Hill, Lloyd, 1990) may be appropriate for a language intended for practical use.

Resolution

Execution of normal logic programs is usually based on unification (Robinson, 1965). A goal is selected and unified with the head of some clause and new substitutions for variables arise. The body of the clause replaces the selected goal and the process continues until it hopefully terminates — and the result can be output as a substitution. This process has been formalized as SLD-resolution (van Emden, Kowalski, 1976, Lloyd, 1987).

The dependencies which arise in the general execution of meta-logical programs are unfortunately more complex than what can be described by substitutions. This is due to the possibility of variables standing for parts of the program currently under execution. Constraint logic programming (Jaffar, Lassez, 1987, Jaffar, Michaylov, 1987) is concerned with logic programs extended with dependencies which cannot be handled by unification, e.g., numerical constraint such as $X > 5$. The strategy for executing constraint logic programs is to split up the state into a substitution and a set constraints which, then, is simplified by a separate machinery. However, the constraint logic methodology does not give a general method for simplifying the constraint nor for deciding whether they actually have a solution.

Our resolution method is a generalization of work done by Martelli and Montanari (1982). They formulate unification as a procedure which transforms equations into a normal form. A normal form corresponds to an idempotent substitution, i.e., no variable in the left hand side of an equation can occur at the right hand side. Each resolution step involves the addition of an equation

selected-goal = *head-of-clause*

to the existing systems of equations and a following application of the transformation algorithm. An advantage of this approach to unification is that it gives a natural way of handling the occur check problem.

In our case, the equations may, furthermore, contain calls of functions which stand for the composition of the syntactic denotation function and a variable renaming function.

Another source of inspiration has been Bonnier's (1989) generalization of the Martelli-Montanari approach to logic programs with calls of externally defined functions, e.g., written in some procedural language. Such function calls will remain unreduced in the systems of equations until their arguments become ground, then they can be reduced into other ground terms.

Relation to logical grammars

The present work has its origin in the author's previous work on adaptable grammars for programming languages (Christiansen, 1985, 1986b, 1988). Using such grammars, the declaration of, say, a variable is described by adding a production rule to the grammar which states the existence of exactly that variable. The intermediate consequence operator defined for DCP in the present paper is actually a restatement in a logical context of the syntactic derivation relation for these grammars. In a similar way, the weak version of SLG-resolution is inherent in a nondeterministic parsing algorithm presented in (Christiansen, 1986b).

In (Christiansen, 1986a) we made a first, ad-hoc, (and semantically insecure!) implementation in Prolog of a subclass of such grammars based on dynamic, local extensions. The close correspondence between the grammars and meta-logic was discovered in an attempt to implement the grammars in a more clean way as a proper extension of Prolog's definite clause grammars (Christiansen, 1989a, 1990b). The paper by Deransart and Maluszynski (1985) which shows a similar relationship between attribute grammars and definite clause programs provided the inspiration for this approach.

Recently, a similar, grammatical concept based on the principle of static, local extensions has been suggested by (Pareschi, Miller, 1990).

3 A syntax for meta-programming in logic

The language of *generative clause programs* generalizes definite clause — or pure Prolog — programs, from which we take the basic syntactic notions. The special constants defined below will be used as ground names for variables.

Definition 3.1. The notions of *constants*, *variables*, *functors*, *terms*, *variants* of a term, and *ground terms* are defined as usual.

For any variable, v, assume a family of constant symbols, called *special constants*,

$$*v, **v, ***v, \ldots$$

□

In this and the following section a general understanding of these concepts is sufficient, however, in section 5 on resolution we will define the notions more carefully.

A usual logical goal represents a statement which is expected to be provable in some program. In a meta-programming environment there are many simultaneous programs involved and therefore we introduce the notion of a meta-goal which consists of a goal together with the program in which it is expected to be true. The symbol "in" is a particular binary functor.

Definition 3.2. A *meta-goal* is a term of the form

$$G \text{ in } P.$$

The subterms G and P are called *goal* and *program parts*, respectively.

A *generative clause* is a term of the form

$$M_0 :- M_1, \ldots, M_n, \quad n \geq 0,$$

where $M_0, \ldots, M_n$ are meta-goals. In the case $n = 0$, a clause is of the form

$$M_0 :- \text{TRUE},$$

where TRUE is a distinguished constant.

A *generative clause program* is a list of generative clauses.

□

We think of TRUE not as a meta-goal which is true by default, rather, it is a marker which indicates the presence of zero meta-goals.

The fact that we view a program as a list of clauses, as opposed to a set, facilitates the representation of programs within themselves. Neither do we do not need to axiomatize separate operators for adding and removing clauses from programs, etc., as is necessary in MetaProlog (Bowen, Weinberg, 1985, Bowen, 1985). The program part on the head of a clause will — in the semantics of GCP to be introduced in the following — be unified with a representation of the current program and thus, we do not need a pseudo-predicate as in MetaProlog, for getting access to it.

On the other hand, this convention implies a strong, syntactic equivalence relation for programs; we will return to this point in section 6.

Note. Program parts in the head of a rule may in principle express strange conditions on the applicability of the rule. Writing, e.g., [–, –, –] would disable a rule in all but three-rule programs. We believe that in any reasonable program, this program part will consist of a variable. However, for simplicity we will not put any syntactic restrictions into the language for this purpose.
□

The relation between certain ground terms and the programs they denote is defined by a function, $\mathcal{D}$, below.†

Definition 3.3. The *syntactic denotation function* for generative clause programs is the function, $\mathcal{D}$, from the set of ground terms to the set of terms defined inductively as follows.

$$\mathcal{D}[\![c]\!] = c, \quad \text{for any non-special constant, } c,$$

$$\mathcal{D}[\![*^n v]\!] = *^{n-1} v, \quad \text{for any variable } v \text{ and } n \geq 1,$$

$$\mathcal{D}[\![f(t_1, \ldots, t_n)]\!] = f(\mathcal{D}[\![t_1]\!], \ldots, \mathcal{D}[\![t_n]\!]).$$

Whenever $\mathcal{D}[\![t]\!] = t'$, we say that *t denotes t'*.
□

Note in the second clause that the right hand side is a variable for $n = 1$, a constant otherwise.

Observation. For any term, there exists a ground term denoting it, i.e., all terms are denotable.
□

For example, the term *x denotes a variable whereas **x denotes a term which denotes a variable. The term,

```
p(*x, *y) in *prog :-
    q(*x) in [(q(**x) in **prog :- *y) | *prog]
```

denotes the following clause; the capital letters indicate "real" variables.

```
p(X, Y) in Prog :-
    q(X) in [(q(*x) in *prog :- Y) | Prog]
```

The meta-goal in the body contains a subterm, which serves as a template for a new rule to extend the current program. If Y is instantiated to, say, "r(*x, **z) in *prog", the template will denote the following rule.

† I.e., instead of using a mapping from terms to the Gödel representation, usually denoted ⌜–⌝, we use a mapping in the other direction.

q(X) in Prog :– r(X, *z) in Prog

Notational shorthands. The goals in a generative clause need not be equipped with explicit program parts. This is not part of the formal definition but is viewed as a notational convention. For any generative clause, the following shorthands apply in the given order:

- If, in a meta-goal, $M_i = G_i$ in P_i, in the body of a clause, the program part, P_i, is identical to the program part for the head of the clause, M_i can be written simply as G_i.
- If the program part in the head of a clause, $M_0 = G_0$ in P_0, is a variable which does not appear anywhere else in the clause, M_0 can be written simply as G_0.

□

This gives a natural injection of definite clause programs into the set of generative clause programs. The following rule, for example,

a :– b, c

is, according to our notational conventions, an abbreviated form of a rule,

a in P :– b in P, c in P.

Our implementation of generative clause programs includes a *where*-notation and a function symbol, &, for combination of grammars, which allow the sample clause above to be written as follows.

```
p(X, Y) in Prog :–
    q(X) in New-prog
where
    New-prog = [New-rule] & Prog,
    New-rule = (q(*x) :– *prog in Y).
```

4 A model-theoretical semantics

Here we define the declarative semantics of generative clause programs as a least Herbrand model. As for definite clause programs, this model can be characterized as the least fixed-point of an immediate consequence operator which is a mapping from Herbrand interpretations to Herbrand interpretations (van Emden, Kowalski, 1976). Our development is quite traditional except that our Herbrand interpretations are concerned with meta-goals instead of with goals and that we cover all possible programs by one and the same model. In the following, we adopt the concepts and notation of (Lloyd, 1987).

The following notion of self-containedness is central for the semantics of generative clause programs. A self-contained clause instance is one which — via the syntactic encoding — is an instance of a pattern found in the program part of its head. The intuitive meaning of this is clarified by an example following the definition.

Definition 4.1. A ground instance of a generative clause,

$$C = \quad (G \text{ in } P) :- B$$

is said to be *self-contained* if $\mathcal{D}[\![P]\!]$ has an instance of the form

$$[\cdots, C, \cdots].$$

□

Example 4.1. There is no circularity problem in this definition. A self-contained clause instance does *not* contain structures isomorphic to itself. What it contains is a representation

of a clause of which it is an instance — and that is something quite different. In this example we will illustrate both the notion of self-containedness and its relation to semantics of the languages to be defined below.

Let $\mathcal{P}_0$ be an abbreviation for the following ground term,

$$[(p(*z) \text{ in } *x :- r \text{ in } *x, q(*z) \text{ in } \mathcal{P}_1), (r \text{ in } *x :- \textsc{True})],$$

where, again, $\mathcal{P}_1$ abbreviates

$$[q(a) \text{ in } **x :- \textsc{True}].$$

$\mathcal{P}_0$ and $\mathcal{P}_1$ are purely textual inventions used in order to make this example readable, we do not introduce any kind of meta-meta-variables or the like.

We will argue informally that the meta-goal $(p(Y) \text{ in } \mathcal{P}_0)$ is true when Y takes the value a. In order to justify this, we see that its program part denotes a program,

$$[(p(Z) \text{ in } X :- r \text{ in } X, q(Z) \text{ in } \mathcal{P}_1'), (r \text{ in } X :- \textsc{True})].$$

where $\mathcal{P}_1'$ abbreviates $[q(a) \text{ in } *x :- \textsc{True}]$. We select the first clause and instantiate Z to a and X to $\mathcal{P}_0$ and we get the — self-contained — clause instance

$$p(a) \text{ in } \mathcal{P}_0 :- r \text{ in } \mathcal{P}_0, q(a) \text{ in } \mathcal{P}_1'.$$

The head of this clause instance is identical to the meta-goal we wanted to prove, so to justify our initial claim, we must prove the meta-goals in the body. The first one is true since its program part denotes a program (the same as above) which has a clause which has the following self-contained instance.

$$r \text{ in } \mathcal{P}_0 :- \textsc{True}$$

The second meta-goal is true since its program part $\mathcal{P}_1'$ denotes a program with one clause,

$$q(a) \text{ in } X :- \textsc{True}.$$

And instantiating X to $\mathcal{P}_1'$ yields a self-contained instance whose head is identical to the meta-goal and whose body is TRUE.

So self-contained clauses are those that arise when a clause is selected in a given meta-goal's program part and its head gets unified the meta-goal.

□

For a definite clause program any instance of a rule can be applied freely in a proof, but in the generative case, we are only interested in those instances that are self-contained. In order to define a suitable notion of Herbrand models for generative programs, there are two possibilities. Either we can define the truth of a generative clause involving this constraint or we can develop the theory for a set of axioms consisting of all self-contained clause instances. For technical reasons we take the latter of these two approaches which otherwise are equivalent.

Definition 4.2. The set of *axioms of generative clause programs*, $\textsc{Axioms}_{\textsc{Gcp}}$, is the set of all self-contained generative clause instances.

An *Herbrand interpretation for generative clause programs* is a set of ground meta-goals.

An *Herbrand model for generative clause programs* is an Herbrand interpretation, $\mathcal{M}$, such that for each

$$M :- M_1, \ldots, M_n \quad \in \textsc{Axioms}_{\textsc{Gcp}},$$

it holds that

$$\{M_1, \ldots, M_n\} \subseteq \mathcal{M} \text{ implies } M \in \mathcal{M}.$$

□

$\text{AXIOMS}_{\text{GCP}}$ is equivalent to an infinite,† definite clause program concerned with a predicate "in" and thus we know from (Lloyd, 87) that the intersection of a non-empty set of Herbrand models itself is an Herbrand model. The intersection of all Herbrand models for generative clause programs is called the *least* Herbrand model and will be denoted $\mathcal{M}^{\text{GCP}}$.

We define the immediate consequence operator for GCP as follows.

Definition 4.3. The mapping T^{GCP} from Herbrand interpretations to Herbrand interpretations is defined as follows.

$$T^{\text{GCP}}(I) = \{M \mid \text{There exists } M :- M_1,\ldots,M_n \in \text{AXIOMS}_{\text{GCP}} \text{ such that } \{M_1,\ldots,M_n\} \subseteq I\}$$

□

T^{GCP} is identical to the usual transformation, T_P, defined for a definite clause program, P, where in this case P is identified with $\text{AXIOMS}_{\text{GCP}}$ as above. Hence we know from (Lloyd, 1987, chapters 1 and 2) that T^{GCP} is continuous and has a least fixed-point, $\text{lfp}(T^{\text{GCP}})$, and finally:

Theorem 4.1. (Fixed-point characterization of the least Herbrand model for generative clause programs).

$$\mathcal{M}^{\text{GCP}} = lfp(T^{\text{GCP}}) = T^{\text{GCP}} \uparrow \omega$$

□

Observation. Let pt_{DCP} denote a definite clause program and pt_{GCP} be the text denoting its injection as a generative clause program. (The injection given by our notational conventions; section 3 above). Then, for all ground goal instances, g,

$$g \in \mathcal{M}_{pt_{\text{DCP}}}$$

if and only if

$$g \text{ in } pt_{\text{GCP}} \in \mathcal{M}^{\text{GCP}},$$

where $\mathcal{M}_{pt_{\text{DCP}}}$ is the usual least Herbrand model for the definite clause program denoted by pt_{DCP}. I.e., DCP constitutes a proper sublanguage of GCP — with respect to the semantic as well as the syntactic aspect.

□

Example 4.2. Let now $\mathcal{P}$ stand for the following non-ground term.

$$[(p(*y) \text{ in } *x :- r \text{ in } *x, Z), (r \text{ in } *x :- \text{TRUE}), (q(a) \text{ in } *x :- \text{TRUE})]$$

The model-based, declarative semantics tells that the meta-goal,

$$p(X) \text{ in } \mathcal{P},$$

is true when, e.g., X takes the value a and Z the value $q(*y)$ in $*x$.

† A definite clause program is usually defined to be a *finite* set of rules. However, it can be checked that the results referenced in this section also hold for infinite programs.

A *complete* resolution method as the one we present below must be imply such computed answers.
□

5 SLG-resolution

The formulation of a model-based semantics for GCP was made using only ground substitutions and always referring to ground instances of clauses. Thus, it was not necessary to consider terms involving the function symbols for the syntactic denotation function — they were reduced away immediately. However, in a complete resolution principle, we have to consider, not only queries with variables but queries with variables in their program parts. In these cases, the syntactic denotation function can be reduced only partially, such that applications to uninstantiated variables will remain. The execution states of the resolution method will have to cope with terms involving such delayed function calls, and the situation is not made less delicate by the fact that these function calls, when their argument becomes ground, return new variables.

We start this chapter by carefully defining the kind of terms and substitutions which are needed.

Section 5.2 defines the notion of constraints and unifiers. The interrelations between the variables involved in an SLG-resolution process are so complicated that we cannot — as in SLD-resolution, (van Emden, Kowalski, 1976, Lloyd, 1987) — use substitutions to record the current set of commitments. Instead, we use evolving constraints which are sets of equations of a certain form. By a unifier for such a system, we mean a substitution which collapses all equations to identities. The resolution process never constructs unifiers, they are used only as a theoretical instrument for correctness considerations.

The operation corresponding to unification is an algorithm which rewrites a constraint into a certain normal form, section 5.3. The resolution method, with selection of clauses and meta-goals supported by not very sophisticated oracles is described in section 5.4. The traditional correctness properties, soundness and completeness, are proved in section 5.5.

In section 5.6 we consider the special case in which no non-ground parts appear — for which our resolution method collapses to a simple interpreter in the style described by Bowen and Kowalski (1982) and studied in more detail by Hill and Lloyd (1989). Of course, the resulting resolution method, which we call weak SLG-resolution, is not complete but it can be implemented quite efficiently and many meta-programming tasks can do without the full completeness.

5.1 A refinement of basic notions

We will assume disjoint sets of
- variables,
- functors with given arity,
- constants, some of which are special constants,
  $*v, **v, ***v, \ldots$ for any variable, v,
- function symbols $\mathcal{R}_i\mathcal{D}[\![-]\!]$ for $i = 0, 1, \ldots$

The function symbol $\mathcal{R}_i\mathcal{D}[\![-]\!]$ stands for the composition of the syntactic denotation function $\mathcal{D}$ and a renaming function for interpretation level i. Having only the combined function symbols relieves us of having to cope with functions that maps variables to other variables (which would result in very strange results when considering the distribution of substitutions over such functions!)

Definition 5.1. The set of *delayed expressions* is defined inductively as follows.

- for any variable, v, and integer i,
  $\mathcal{R}_i\mathcal{D}[\![v]\!]$
  is a delayed expression.
- for any delayed expression, d, and integer i,
  $\mathcal{R}_i\mathcal{D}[\![d]\!]$
  is a delayed expression.

The set of *generalized terms* is defined inductively as follows.

- any variable, v, is a generalized term,
- any constant, a, is a generalized term,
- any delayed expression is a generalized term,
- whenever f is a functor of arity n and $t_1, \ldots, t_n$ are generalized terms, the structure
  $f(t_1, \ldots, t_n)$
  is a generalized term.

A *term* is a generalized term without delayed expressions, a *ground* term is a term without variables. A *structure* is a generalized term which starts with a functor or which is a constant.
□

Example 5.1. The expression $\mathcal{R}_7\mathcal{D}[\![f(a,x)]\!]$ is not a generalized term. However, the definition below will evaluate it into the generalized term $f(a, \mathcal{R}_7\mathcal{D}[\![x]\!])$, assuming a is a constant and x a variable.
□

In the following, we define the operational meaning of the $\mathcal{R}_i\mathcal{D}[\![-]\!]$ function symbols. By "operational" we mean that its behavior is defined also when applied to non-ground terms. We will assume functions $\mathcal{R}_i[\![-]\!]$ which maps variables to new, unique variables. It is assumed that the sets of variables produced by each such function are disjoint. Without otherwise mentioning, we will also assume that these variables are disjoint from those that can appear in queries given by the user and those below referred to as "new" variables.

Definition 5.2. The functions $\mathcal{R}_i\mathcal{D}[\![-]\!], i = 0, 1, \ldots$ from terms to generalized terms are defined inductively as follows.

- $\mathcal{R}_i\mathcal{D}[\![a]\!] = a$ for any non-special constant a,
- $\mathcal{R}_i\mathcal{D}[\![*^n v]\!] = *^{n-1} v, \quad n > 1$ for any special constant, $*^n v$,
- $\mathcal{R}_i\mathcal{D}[\![*v]\!] = \mathcal{R}_i[\![v]\!]$ for any special constant of the form $*v$,
- $\mathcal{R}_i\mathcal{D}[\![e]\!]$ is the delayed expression $\mathcal{R}_i\mathcal{D}[\![e]\!]$ for any variable or delayed expression e,
- $\mathcal{R}_i\mathcal{D}[\![f(t_1, \ldots, t_n)]\!] = f(\mathcal{R}_i\mathcal{D}[\![t_1]\!], \ldots \mathcal{R}_i\mathcal{D}[\![t_n]\!])$ for any functor of arity n and terms $t_1, \ldots, t_n$.

For any expression, e of the form $\mathcal{R}_i\mathcal{D}[\![t]\!]$, the *value* of e is the generalized term which results from applying $\mathcal{R}_i\mathcal{D}[\![-]\!]$ to t.
□

We will motivate our definitions below concerned with substitutions by the following example.

Example 5.2. The $\mathcal{R}_i\mathcal{D}[\![-]\!]$ functions are unusual in the sense that they return variables and as a consequence, substitutions behave slightly different from what we may expect. In

order for the notion of substitutions to support any intuitive understanding, we must require associativity with respect to composition. I.e.,

$$(t\theta)\varphi = t(\theta\varphi)$$

We also need a convention for distributing substitutions over function symbols. For the purely syntactic functors, we must expect the normal distribution rule, but can we expect the following one for our strange functions?

$$\mathcal{R}_i\mathcal{D}[\![t]\!]\theta \stackrel{?}{=} \mathcal{R}_i\mathcal{D}[\![t\theta]\!]$$

The answer is "no" which we will see from the following example. Let $\theta = \{x \rightarrow {*a}\}$ such that the value of $\mathcal{R}_7\mathcal{D}[\![*a]\!]$ is a variable A_7 and let $\theta = \{A_7 \rightarrow const\}$. So according to the associativity property, we should have

$$(\mathcal{R}_7\mathcal{D}[\![x]\!]\theta)\varphi = \mathcal{R}_7\mathcal{D}[\![x]\!](\theta\varphi).$$

Let us process the left hand side according to the questioned distribution rule . . .

$$(\mathcal{R}_7\mathcal{D}[\![x]\!]\theta)\varphi \stackrel{?}{=} (\mathcal{R}_7\mathcal{D}[\![x\theta]\!])\varphi = (\mathcal{R}_7\mathcal{D}[\![*a]\!])\varphi = A_7\varphi = const$$

. . . and now the right hand side,

$$\mathcal{R}_7\mathcal{D}[\![x]\!](\theta\varphi) \stackrel{?}{=} \mathcal{R}_7\mathcal{D}[\![x\theta\varphi]\!] = \mathcal{R}_7\mathcal{D}[\![*a\varphi]\!] = \mathcal{R}_7\mathcal{D}[\![*a]\!] = A_7!!!$$

The problem is with this distribution rule, that a substitution can be moved from a variable (returned by one of our functions) over to a special constant (giving rise to the variable) for which it has no effect. However, do not despair, the following distribution rule works.†

$$\mathcal{R}_i\mathcal{D}[\![t]\!]\theta = \mathcal{R}_i\mathcal{D}[\![t\theta]\!]\theta$$

And in order to use this as a meaningful reduction rule, we must also require idempotency with respect to composition, i.e., $\theta\theta = \theta$.
□

We emphasize that the model-based semantic definition of section 4 only applies substitutions which are ground, and ground substitutions are special cases of we define in the following.

Definition 5.3. For any mapping, m, from a set of variables to terms, let dom(m) be the set of variables for which m defines a value and let range(m) be the set of variables which can occur in a term $m(v)$ for $v \in$ dom(m); vars(m) is the union of dom(m) and range(m).

A *substitution* is a mapping θ from a finite set of variables to terms, such that dom(θ) $\cap$ range(θ) = $\emptyset$.

A *ground* substitution is a substitution, γ, such that $\gamma(v)$ is a ground term for all $v \in$ dom(γ).

Whenever V is a set of variables, the *restriction* of a substitution θ to V, denoted $\theta|_V$, is the substitution which agrees with θ for any variable in V and which is undefined for any other variable.

A substitution, η, is a *renaming of variables* if, for any two distinct variables, $v_1, v_2 \in$ dom(η), that $\eta(v_1)$ and $\eta(v_2)$ are distinct variables.
□

Definition 5.4. The *value* of a generalized term, t, under a substitution, θ, is denoted $t\theta$ and is defined inductively as follows.

- $v\theta = \theta(v)$ for a variable $v \in$ dom(θ), otherwise $v\theta = v$,
- $f(t_1,\ldots,t_n)\theta = f(t_1\theta,\ldots,t_n\theta)$ for any term of the form $f(t_1,\ldots,t_n)$,

† It is a good exercise for the reader to redo the example with the correct rule.

– $\mathcal{R}_i\mathcal{D}[\![e]\!]\theta = t\theta$ where t is the value of $\mathcal{R}_i\mathcal{D}[\![e\theta]\!]$ for any delayed expression or variable e for which $e\theta$ is ground.

□

Discussion of definition 5.4. The value of $\mathcal{R}_i\mathcal{D}[\![t]\!]\theta$ is undefined whenever $t\theta$ is non-ground. There is no conceptual reason for this, it is simply a matter of convenience. The definitions can be extended to handle these cases by the introduction of yet another level of generalized terms which include delayed substitutions. As defined above, the value of a generalized term under a substitution, provided it is defined, is always a term, i.e., an expression without $\mathcal{R}_i\mathcal{D}[\![-]\!]$ function symbols. In this paper, we do not need this generalization (and the additional complication!) since our substitutions always are chosen such that we avoid the problem. Also, by nature of idempotent substitutions, composition is not always defined (see, e.g., Ko, Nadel, 1991).

□

Definition 5.5. The *composition* of two substitutions, θ_1 and θ_2, is the mapping $\theta_1\theta_2$ defined as follows. However, if $\theta_1\theta_2$ is not a substitution, the composition is undefined.

For any $v \in \mathrm{dom}(\theta_1) \cup \mathrm{dom}(\theta_2)$ let

– $(\theta_1\theta_2)(v) = s\theta_2$ provided $\theta_1(v) = s$, otherwise

– $(\theta_1\theta_2)(v) = \theta_2(v)$,

except if $(\theta_1\theta_2)(x)$ defined as above equals x for a variable x, we let $(\theta_1\theta_2)(x)$ be undefined.

□

Proposition 5.1. Substitutions are idempotent with respect to composition, i.e., for any substitution, θ, the following holds.

$$\theta = \theta\theta$$

□

Proof. Follows from the definition of substitution, for any variable $v \in \mathrm{dom}(\theta)$, θ does not affect any variable in $\theta(v)$.

□

Proposition 5.2. Let ε be the substitution for which $\mathrm{dom}(\varepsilon)$ is empty. Then, for any substitution θ the following holds.

$$\theta\varepsilon = \varepsilon\theta = \theta$$

□

The proof is trivial.

Proposition 5.3. Let t be any generalized term and θ_1 and θ_2 any substitutions. If both $(t\theta_1)\theta_2$ and $t(\theta_1\theta_2)$ are defined, it holds that

$$(t\theta_1)\theta_2 = t(\theta_1\theta_2).$$

□

Proof. We assume that both $(t\theta_1)\theta_2$ and $t(\theta_1\theta_2)$ are defined. First, let t be any (non-generalized!) term. We use induction over the structure of t.

– t is a constant, a. Trivial: $(a\theta_1)\theta_2 = a\theta_2 = a = a(\theta_1\theta_2)$.

- $t = f(t_1, \ldots, t_n)$. Follows by induction.
- t is a variable v.
  - if $\theta_1(v) = s$ then $(v\theta_1)\theta_2 = s\theta_2 = (\theta_1\theta_2)v$,
  - otherwise $(v\theta_1)\theta_2 = v\theta_2 = (\theta_1\theta_2)v$.

We consider, now, the case where t is a delayed expression. Firstly, let $t = \mathcal{R}_i\mathcal{D}[\![v]\!]$ where v is a variable. We observe that $v(\theta_1\theta_2) = (v\theta_1)\theta_2 = v\theta_1$ since $v\theta_1$ must be ground and we can use a case proved above. Now,

$$\mathcal{R}_i\mathcal{D}[\![v]\!](\theta_1\theta_2) = \mathcal{R}_i\mathcal{D}[\![v(\theta_1\theta_2)]\!](\theta_1\theta_2) = \mathcal{R}_i\mathcal{D}[\![v\theta_1]\!](\theta_1\theta_2),$$

and since $\mathcal{R}_i\mathcal{D}[\![v\theta_1]\!]$ is a term, we can continue using the already proved part of the proposition,

$$= (\mathcal{R}_i\mathcal{D}[\![v\theta_1]\!]\theta_1)\theta_2 = (\mathcal{R}_i\mathcal{D}[\![v]\!]\theta_1)\theta_2,$$

which finishes this part of the proof; the final equality is an application of definition 5.4.

Let now $t = \mathcal{R}_i\mathcal{D}[\![d]\!]$ where d is a delayed expression for which the proposition holds by induction hypothesis, i.e., $d(\theta_1\theta_2) = (d\theta_1)\theta_2 = d\theta_1$ since $d\theta_1$ must be ground. So the argument is similar to the previous case.

The final case where t is a structure with delayed subexpressions follows by induction referring to the cases already proved.

□

Proposition 5.4. If t is a generalized term and γ grounds any variable in t which is enclosed in $\mathcal{R}_i\mathcal{D}[\![-]\!]$ function symbols, then both $t(\gamma\sigma)$ and $(t\gamma)\sigma$ are defined and equal for any substitution σ. If $(t\theta_1)\theta_2$ is defined and the composition $\theta_1\theta_2$ is defined, then $t(\theta_1\theta_2)$ is defined.

□

This proposition guarantees that the kind of substitutions, unifiers and solutions, to be defined later always yields a value for the objects to which it is applied.

Proof. Follows immediately from the definitions and proposition 5.3.

□

Proposition 5.5. Let θ_1, θ_2, and θ_3 be substitutions. Then if both $(\theta_1\theta_2)\theta_3$ and $\theta_1(\theta_2\theta_3)$ are defined, then

$$(\theta_1\theta_2)\theta_3 = \theta_1(\theta_2\theta_3).$$

□

Proof. We prove for any variable, v, that $v((\theta_1\theta_2)\theta_3) = v(\theta_1(\theta_2\theta_3))$. This follows by successive applications of proposition 5.3.

$$v((\theta_1\theta_2)\theta_3) = (v(\theta_1\theta_2))\theta_3 = ((v\theta_1)\theta_2)\theta_3 = (v\theta_1)(\theta_2\theta_3) = v(\theta_1(\theta_2\theta_3))$$

□

Definition 5.6. Whenever, for substitutions σ_1 and σ_2, that $\sigma_2 = \sigma_1\varphi$ for some substitution φ, we say that σ_2 is a *specialization* of σ_1. If, furthermore, $\mathrm{vars}(\sigma_1) \cap \mathrm{dom}(\varphi) = \emptyset$, we call σ_2 an *extension* of σ_1 and σ_1 a *restriction* of σ_2.

Whenever Σ and Σ' are sets of substitutions such that
- for any $\sigma \in \Sigma$ there is a $\sigma' \in \Sigma'$ such that σ' is an extension of σ, and
- for any $\sigma' \in \Sigma'$ there is a $\sigma \in \Sigma$ such that σ is a restriction of σ',

we say that Σ' is an *extension* of Σ — and that Σ is a *restriction* of Σ'.

□

Intuitively, σ_2 being an extension of σ_1 means that σ_2 can take care of new variables, but the existing bindings of σ_1 cannot be modified.

The relationship extension/restriction between two sets of unifiers is a strong condition. If Σ is a restriction of Σ', all information in Σ is present and undistorted in Σ'. The other way round, Σ contains all information about a particular set of variables which can be squeezed out of Σ'.

5.2 Constraints and their unifiers

Definition 5.7. A *constraint* is a set of equations, each one of the form $l = r$ where l and r are generalized terms.

Whenever C is a constraint and θ a substitution, $C\theta$ is the constraint which appears when each equation, $l = r$, is replaced by the $val_l = val_r$, where val_l and val_r are the values of, respectively, $l\sigma$ and $r\sigma$.

A *unifier* for a constraint C is a substitution, σ, such that for any θ such that $C\sigma\theta$ is defined, that $C\sigma\theta$ consists of equations between identical terms. The notation su(C) will refer to the set of all unifiers for C. Two constraints, C_1 and C_2, are considered *equivalent* if su(C_1) = su(C_2).
□

Our definition of a unifier makes it possible to consider as unifiers also substitutions which do not ground the arguments to calls of $\mathcal{R}_i\mathcal{D}[\![-]\!]$ functions. So, for example, any substitution is a unifier for the constraint $\{\mathcal{R}_0\mathcal{D}[\![x]\!] = \mathcal{R}_0\mathcal{D}[\![x]\!]\}$. If we had been less careful in the definition, only substitutions which grounded x would be unifiers — and thus the constraint would not be equivalent with the empty constraint (!).

In most cases, the statement "for any θ for which $t\sigma\theta$ and $t'\sigma\theta$ are defined, $t\sigma\theta = t'\sigma\theta$" is equivalent to saying "$t\sigma = t'\sigma$"; the exceptions are when t and t' have variables within $\mathcal{R}_i\mathcal{D}[\![-]\!]$ function symbols which are not grounded by σ. For brevity, we may occasionally write "$t\sigma = t'\sigma$" when we really mean the more precise equivalence statement.

Example 5.3. Consider the following constraint,

$$C = \{\mathcal{R}_7\mathcal{D}[\![v]\!] = (x :\!- y)\}.$$

The substitutions, σ_1 and σ_2, defined below are both unifiers for C.

$$\sigma_1 = \{v \rightarrow {*a}, w_a \rightarrow (x :\!- y)\}$$

$$\sigma_2 = \{v \rightarrow ({*a} :\!- {*b}), w_a \rightarrow x, w_b \rightarrow y\}$$

It is assumed that w_a and w_b are the variables returned by $\mathcal{R}_7\mathcal{D}[\![{*a}]\!]$ and $\mathcal{R}_7\mathcal{D}[\![{*b}]\!]$. These unifiers are examples of what we can consider as "useful" unifiers contained in the constraint. A given constraint will in general contain many, also weird, unifiers. This should not be a surprise since a query of the form "give me a program such that a query such-and-such is provable" clearly can be answered in many strange ways — so part of a methodology for using SLG-resolution for program generation would be to set up requirements in the initial query of what should be considered a useful program for the given application.
□

Constraints are used in our resolution method to record the current set of commitments. The interesting question to ask is whether or not a given constraint actually has unifiers. As for plain logic programs, a constraint which implies an object occur in itself may not have a solution. Since we have (non-syntactic) function symbols in our constraint we must distinguish between different kinds of (syntactic) occurrences.

Definition 5.8. The notion of a generalized term *occurring* in another and when the occurrence is *serious* is defined as follows.

- For any generalized term t, t occurs in t.
- A generalized term t occurs seriously in $f(\cdots t \cdots)$, f a functor.
- A generalized term t occurs in $\mathcal{R}_i\mathcal{D}[\![t']\!]$ if t occurs in t'.

□

Example 5.4. Consider the following two equations which have a serious, resp. a non-serious, occurrence of x, resp. y, on their right, resp. left, hand sides.

$$x = f(\mathcal{R}_2\mathcal{D}[\![x]\!])$$

$$\mathcal{R}_2\mathcal{D}[\![y]\!] = y$$

The first equation cannot have a unifier since any value for x would yield a right hand side with one more instance of "f" than the left hand side. In the second equation, we can, e.g., let y be $*a$ where $\mathcal{R}_2\mathcal{D}[\![*a]\!]$ evaluates to a variable w_a and let the value of w_a be the same as the value of y, i.e., $*a$. We may also put in any non-special constant for y or a "weird" structure such as $f(*a, g(*b, c))$ and get unifiers for the equation.

The following equation will have unifiers despite the fact that a common subexpression occurs in syntactically different positions on the left and right hand sides.

$$\mathcal{R}_7\mathcal{D}[\![\mathcal{R}_9\mathcal{D}[\![x]\!]\,]\!] = f(\mathcal{R}_9\mathcal{D}[\![x]\!])$$

This is an example of an equation $l = r$ for which $l < r$ where $<$ refers to the ordering defined below.

□

We will now introduce a notion of constraints in normal form. This is interesting since such constraints are guaranteed to have unifiers and we can present an algorithm which will convert any constraint with has unifiers into this form — or reject it if it has no unifier. For this purpose, we introduce an ordering on generalized terms.

Definition 5.9. The partial ordering, $<$, on generalized terms is defined as follows.

- For any delayed expression, d, and variable, v, $d < v$,
- For any delayed expression, d, and structure, s, $d < s$,
- For any variable, v, and structure, s, $v < s$.
- For any two distinct variables, v_1 and v_2, assume some standard ordering such that one and only one of $v_1 < v_2$ and $v_2 < v_1$ holds.
- For any two distinct, delayed expressions, d_1 and d_2,
  - if d_2 occurs in d_1, let $d_1 < d_2$; otherwise,
  - if none of d_1 and d_2 occur in the other, assume both $d_1 < d_2$ and $d_2 < d_1$.

□

Definition 5.10. A constraint, N, is in *normal form* (or *normalized*) if

$$N = \{l_1 = r_1, \ldots, l_n = r_n\}$$

where $l_1, \cdots, l_n, r_1, \cdots r_n$ are generalized terms with, for all i, $l_i < r_i$, and each l_i occurs only once in N.

□

Example 5.5. Consider the following constraint.

$$\{(x_1 :- x_2) = (\mathcal{R}_{17}\mathcal{D}[\![x_3]\!] \text{ in } \mathcal{R}_{17}\mathcal{D}[\![x_4]\!] :- \mathcal{R}_{17}\mathcal{D}[\![x_5]\!]),$$
$$\mathcal{R}_{17}\mathcal{D}[\![x_3]\!] = \mathcal{R}_5\mathcal{D}[\![\mathcal{R}_{17}\mathcal{D}[\![x_3]\!]]\!]\}$$

The following, normalized constraint is an equivalent normal form.

$$\{x_1 = (\mathcal{R}_{17}\mathcal{D}[\![x_3]\!] \text{ in } \mathcal{R}_{17}\mathcal{D}[\![x_4]\!]),$$
$$\mathcal{R}_{17}\mathcal{D}[\![x_5]\!] = x_2,$$
$$\mathcal{R}_5\mathcal{D}[\![\mathcal{R}_{17}\mathcal{D}[\![x_3]\!]]\!] = \mathcal{R}_{17}\mathcal{D}[\![x_3]\!]\}$$

This is a normal form which can be produced by the normalization algorithm to be presented below. Note that the normal form condition implies for any equation $l < r$, that l cannot occur in r, but r can have a non-serious occurrence in l.
□

Lemma 5.1. A constraint in normal form has at least one unifier.
□

Proof. Let N be a constraint in normal form. We will construct a unifier for N as follows. Let α_0 be the empty substitution and let $N_0 = N$. If N_{j-1} contains a delayed expression $\mathcal{R}_{i_j}\mathcal{D}[\![x_j]\!]$ where x_j is a variable, let $\alpha_j = \alpha_{j-1}\{x_j \rightarrow *a_j\}$ where $*a_j$ is a special constant chosen such that the variable given by $\mathcal{R}_{i_j}\mathcal{D}[\![*a]\!]$ does not occur in N_{j-1}. Construct N_j from N_{j-1} replacing any occurrence of $\mathcal{R}_{i_j}\mathcal{D}[\![x_j]\!]$ by the variable given by $\mathcal{R}_{i_j}\mathcal{D}[\![*a]\!]$. Let k be largest k such that α_k is defined in this way. The constraint $N\alpha_k = N_k$ will contain only equations of the form

$v = t$, v a variable and t a term,

where each such v occurs only once. Let β be the substitution such that $\beta(v) = t$ for any $(v = t) \in N\alpha_k$ and β undefined for any other variable. Clearly β is a unifier for N_k and thus $\alpha_k\beta$ is a unifier for N.
□

5.3 Unification

The current set of commitments made in a so far successful resolution process is represented by a constraint in normal form. A resolution step will add an equation

head-of-selected-clause = selected-meta-goal

which represents the possible unification of its left and right hand sides. The normalization algorithm will, if possible, bring the constraint back into normal form and thus it serves as a normalization algorithm.

Our algorithm has been adapted from (Martelli, Montanari, 1982). The algorithm takes as input a constraint, C, and produces an equivalent constraint in normal form — or failure if there is no such normal form. In the following, a new variable means a variable which does not occur in the (more or less implicit) context of constraints and other objects which may contain variables.

Normalization algorithm. Initially, let $C_0 = C$ and $i = 0$. Repeatedly do the following: Select any equation $l = r$ in C_i such that one of the rules 1 to 4 applies. If no such equation exists then stop with C_i as result. Otherwise perform the corresponding action, i.e., stop with failure or construct C_{i+1} from C_i. Then increment i by 1.

1) If l and r are identical, remove the equation $l = r$.
2) $l < r$ and l which occurs elsewhere in C_i. If l occurs in r then stop with failure, otherwise replace all other occurrences of l in C_i by r (i.e., leave $l = r$ unchanged). If any expression of the form $e = \mathcal{R}_i\mathcal{D}[[t]]$, where t is neither a variable or a delayed expression, evaluate e and replace it by its value.
3) If $r < l$ and not $l < r$, replace $l = r$ by $r = l$.
4) $l = f_1(l_1, \ldots, l_n)$ and $r = f_2(r_1, \ldots, r_m)$ where f_1 and f_2 are functors. If f_1 and f_2 are different or $n \neq m$ then stop with failure, otherwise replace $l = r$ by $l_1 = r_1, \ldots, l_n = r_n$.

□

Example 5.6. We will illustrate the algorithm's use of the $<$ ordering. Consider an equation such as

$$x = \mathcal{R}_{27}\mathcal{D}[[\mathcal{R}_7\mathcal{D}[[x]]]].$$

It does not satisfy the $l < r$ condition and it has to be turned around by step 3 before step two can apply. Now all other occurrences of $\mathcal{R}_{27}\mathcal{D}[[\mathcal{R}_7\mathcal{D}[[x]]]]$ can be replaced by the simpler term x. In its treatment of equations between variables and structures, our algorithm works exactly as Martelli and Montanari's original algorithm.

□

In order to prove the correctness of the algorithm, we will prove the following.
- For any possible step which transforms a constraint C_i into C_{i+1}, C_i and C_{i+1} are equivalent.
- The result is in normal form.
- The algorithm will terminate.

Proposition 5.6. Each possible transformation step defined by the rules 1 to 4 has the property that if it changes C_i into C_{i+1}, C_i and C_{i+1} are equivalent. In case a step returns failure for a constraint, C_i, there exists no unifier for C_i.

□

Proof. Step 1 removes equations which are satisfied under any substitution, so any unifier for C_i is a unifier for C_{i+1} and vice-versa. Step 2 exchanges generalized terms l with generalized terms r which are indifferent under any unifier, and thus, does not affect the unifier set. Steps 3 and 4 clearly does not affect the unifier set either.

If the algorithm returns failure, then step 2 has found an equation of the form

$$e = f(\cdots e \cdots)$$

where e is either a variable or a delayed expression, or step 4 has found an equation

$$f(\cdots) = g(\cdots).$$

In none of the cases, a unifier can exist.

□

Proposition 5.7. Assume the algorithm is given a constraint C as input and it outputs a constraint N. Then N is an equivalent normal form for C. If the algorithm outputs failure, no equivalent normal form exists.

□

Proof. If the algorithm stops with failure, induction using proposition 5.6 gives that C has no unifier and hence, by lemma 5.1, that it has no normal form.

Assume now, that the algorithm stops outputting the constraint N. If N is in normal form, then induction using proposition 5.6 gives that N is an equivalent normal form for C.

Now we show that N actually is of normal form. The proof will be indirect, so we assume that N is *not* in normal form. This implies that N has an equation $l = r$ where either not $l < r$ or l occurs somewhere else in N. If not $l < r$ then either $r < l$, which is not possible since in this case step 3 could have been applied, or l and r are not comparable by $<$. The only possibility for this is if both l and r are structures. This is not the case either, since step 4 or perhaps step 1 could have been applied. Is it possible, then, that l occurs somewhere else in N? No, because then step 2 could have been applied.

Hence it is not the case that N is not normal.

□

Proposition 5.8. Given a constraint, C, as input, the normalization algorithm terminates with either a constraint or failure.

□

The proof has not been found yet — so proposition 5.8 still remains a conjecture.

Lemma 5.2. Correctness of the normalization algorithm.
Whenever C is a constraint, the normalization algorithm will output an equivalent normal form N if and only if such a normal form exists. Otherwise it outputs failure.

□

Proof. Follows from propositions 5.7 and 5.8.

□

We end this subsection with a stronger form of lemma 5.1.

Lemma 5.3. A constraint C has a unifier if and only if it has an equivalent normal form.

□

Proof. Follows from lemmas 5.1 and 5.2.

□

5.4 The resolution procedure

In this section, we give the definitions concerned with SLG-resolution. For the reason of clarity, all considerations about correctness are referred to section 5.5. We remind that $\mathcal{M}^{\text{GCP}}$ is the least Herbrand model for GCP, cf. section 5.

Definition 5.11. What is understood by a *query* is defined inductively as follows.

- A meta-goal is a query.
- A variable is a query.
- A delayed expression is a query.
- The constant TRUE is a query.
- Whenever A and B are queries, the structure (A, B) is a query.

A query is *true* if it is a sequence of meta-goals $\in \mathcal{M}^{\text{GCP}}$ or occurrences of the constant TRUE.

A *resolution state* or, for short, a *state*, is a normalized constraint, $S = C \cup \{\text{QUERY} = Q\}$, where QUERY is a distinguished variable and Q a query.

An *initial* state is a resolution state of the form $\{\text{QUERY} = Q\}$ where Q is a query which is a sequence of meta-goals in which no delayed expressions occurs. A *success* state is a resolution state of the form $C \cup \{\text{QUERY} = \text{TRUE}, \ldots, \text{TRUE}\}$.

□

It is inherent in the definition that a resolution state, considered as a constraint, always has at least one unifier, but not necessarily that there exists a substitution which maps the query part into a true query.

Example 5.7. We will illustrate the slight difference between variables and delayed expressions appearing in queries. Consider a query,

$$\cdots, x, \cdots, \mathcal{R}_7\mathcal{D}[\![y]\!], \cdots.$$

It may arise from an initial query of the form

$$p \text{ in } [\cdots, (p \text{ in } *c :- \cdots, *a, \cdots, y, \cdots), \cdots].$$

The occurrence of y means that the user wants a value for y which represents a piece of program text. The sloppy use of $*a$, which denotes a variable x, in the position of a sub-goal means that (each renamed version of) x must, for each call of the clause, be instantiated to a meta-goal (not necessarily the same one) in order for the query to succeed, but the user is not interested in those meta-goals. This corresponds to the fact that most Prolog interpreters accept clauses such as $p :- \cdots, X, \cdots$ if X is instantiated to a goal whenever reached by execution.

The resolution method and normalization algorithm may also exchange a delayed expression in a query with variables.
□

In normal SLD-resolution, it is straightforward to select a goal and a clause, since the selection always is made from completely known collections. In our case, we may have delayed expressions or variables in our queries which stand for bodies of unknown clauses. If, for example, V is a variable in the current query, it may be the case that it really should stand for a clause body with five (yet unknown) meta-goals — and (sooner or later) the third one should be selected.

In order to handle this, we have added simple-minded oracles to the selection step which will provide the possibly missing structure.† The oracle's decisions are recorded as equations which subsequently can be taken into the normalization process. The oracle may need to require new variables. By a new variable, we mean a variable which does not occur in the implicit context of constraints and queries — and also in the substitutions which are used in the correctness proofs in the subsequent section. Note also that we are very careful in formalizing what it means to "replace the selected meta-goal by something", hence the notion of intermediate query and replacement variable.

Definition 5.12. We say that a meta-goal M is *selected* from a query Q giving an intermediate query, Q', with replacement position v_R and additional equations E if the property $\text{select}(M, Q, Q', E)$ defined below holds.

For a meta-goal M, queries Q and Q', variable r_R, and set of equations E, the property

$$\text{select}(M, Q, Q', v_R, E)$$

is defined inductively as follows.

1) $\text{select}(M, M, v_R, v_R, \{\,\})$
   if M is a meta-goal; v_R is a new variable.

2) $\text{select}(M, (A, B), (A', B), v_R, E)$
   if $\text{select}(M, A, A', v_R, E)$, where A and B are queries.

† The nondeterminism in the oracles' decisions about missing structure may annoy some readers, so we will comment on it below the definitions.

3) $\mathrm{select}(M,(A,B),(A,B'),v_R,E)$
if $\mathrm{select}(M,B,B',v_R,E)$ where A and B are queries.

4) $\mathrm{select}(M,v,Q',v_R,\{v=(v_A,v_B)\}\cup E)$
if $\mathrm{select}(M,(v_A,v_B),Q',v_R,E)$, where v is a variable, v_A and v_B new variables.

5) $\mathrm{select}((v_g \text{ in } v_p),v,v_R,v_R,\{v=(v_g \text{ in } v_p)\})$
where v is a variable, v_R, v_g and v_p new variables.

6) $\mathrm{select}(M,d,Q',v_R,\{v=d\})$
if $\mathrm{select}(M,v,Q',v_R,E)$, where d is a delayed expression, v a new variable.

□

Example 5.8. Given a query $(g \text{ in } p,X)$, the selection procedure may produce a selected query with new variables, $(v_g \text{ in } v_p)$, an intermediate query $(g \text{ in } p,v_1,v_R)$ where v_1 is some new variable and v_R the replacement variable. The set of additional equations is $\{X=(v_1,v_2),\ v_2=(v_g \text{ in } v_p)\}$.

So in this case, the oracle has decided that X is a sequence of two elements where the second one is the selected goal.

□

Now, having selected a meta-goal, the next step will be to apply an $\mathcal{R},\mathcal{D}[\![-]\!]$ function to its perhaps non-ground program part. How this can be done is described in definition 5.2 above. The resulting (perhaps generalized!) term is then supposed to be a list of — supposed — clauses one of which must be selected for the current resolution step.

Definition 5.13. We say that a clause, cl, is *selected* from a term, t, with additional equations, E, if property

$\mathrm{member}(cl,t,E)$

defined below holds.

For any clause cl, generalized term t, and sets of equations, E, the property $\mathrm{member}(cl,t,E)$ is defined inductively as follows.

1) $\mathrm{member}((H :- B),[(H' :- B')|t],E)$
if E is the least set of equations such that
– $H=H'$ is a meta-goal, or
– H' is a variable v_h, $H=(v_g \text{ in } v_p)$, v_g and v_p new variables, $(v_h=(v_g \text{ in } v_p))\in E$,

and

– $B=B'$, or
– B' is a variable v_b and $B=\textsc{True}$ and $(v_b=\textsc{True})\in E$, or
– B' is a delayed expression d and $B=\textsc{True}$ and $(\textsc{True}=d)\in E$.

2) $\mathrm{member}(cl,[v|t],\{v=(v_h :- v_b)\}\cup E)$
if $\mathrm{member}(cl,[v_h :- v_b],E)$ where v is a variable, v_h and v_b new variables.

3) $\mathrm{member}(cl,[d|t],\{v=d\}\cup E)$
if $\mathrm{member}(cl,[v],E)$ where d is a delayed expression, v a new variable.

4) $\mathrm{member}(cl,[t_1|t_2],E)$
if $\mathrm{member}(cl,t_2,E)$ where t_1 and t_2 are arbitrary, generalized terms.

5) $\mathrm{member}(cl,v,\{v=[v_1|v_2]\}\cup E)$
if $\mathrm{member}(cl,[v_1|v_2],E)$ where v is a variable, v_1, v_2 new variables.

6) member$(cl, d, \{v = d\} \cup E)$
if member(cl, v, E) where d is a delayed expression, v is a new variable.

□

Note, in the first clause, that the case "$B = B'$" may apply for all possible kinds of B' — also those for which the two other cases might apply. The oracle built into this definition may thus decide at this point that an unknown clause can be either a rule or a fact. It also sets up a structure for the head of the clause if it does not exist already. In case of a completely unknown program, the oracle can generate a program of any length with the selected clause being in any position. If the term t is a structure which cannot be a program, the membership property is not defined.

Example 5.9. Given a term $[\mathcal{R}_7\mathcal{D}[[x]]]$, the membership procedure may return a clause with new variables, $(v_g \text{ in } v_p) :- v_b$ and a set of additional equations $\{v_1 = \mathcal{R}_7\mathcal{D}[[x]], v_1 = (v_h :- v_b), v_h = (v_g \text{ in } v_p)\}$.

So the procedure was given a program consisting of one unknown element and the oracle decreed it be a clause — giving only the most rudimentary structure needed in order to recognize it as a clause.

□

The nondeterminism in the oracles' ways of providing missing structure is impractical in an implementation since it may require an immense amount of searching before the right decision is made. But it should be made clear that the possible outputs always are numerable, so the total output made by the (at most) countable number of oracles in some proof tree skeleton is also numerable — and thus in principle implementable. An actual implementation should, of course, be worked out more carefully than just coding the selection procedures literally according to our definitions. We prefer to think of our definitions above as formally convenient abstractions over some finite representation for facts like "the value of v is a list with $(g \text{ in } p)$ as an element." This discussion touches also some aspects about program equivalence which we will return to in the final section.

Definition 5.14. A state, S', is *derived from* a state $S = C \cup \{\text{QUERY} = Q\}$ at level i, $i = 0, 1, 2 \ldots$, if

- select$((g \text{ in } p), Q, Q', v_R, E_1)$,
- member$((H :- B), \mathcal{R}_i\mathcal{D}[[p]], E_2)$, and
- the normal form algorithm outputs S' when applied to the constraint

$$C \cup E_1 \cup E_2 \cup \{(g \text{ in } p) = H, v_R = B\} \cup \{\text{QUERY} = Q'\}.$$

□

Note that the normalization algorithm automatically will put in the body of the selected clause, B, in the intermediate query, Q', in the position of the replacement variable, v_R.

Definition 5.15. An SLG-*computation* consists of a series of resolution states,

$$S_0, S_1, S_2, \ldots,$$

such that S_0 is an initial resolution state and S_{i+1} is derived from S_i at level i.

An SLG-*resolution* of a query, Q, is a finite SLG-computation whose initial state is

$$\{\text{QUERY} = Q\}$$

and which ends in a success state $C \cup \{\text{QUERY} = \text{TRUE}, \ldots, \text{TRUE}\}$. In this case, C is a *success constraint* for Q.

□

The output from a successful resolution is thus a constraint in normal form, N, and not, as usual, a substitution; substitutions are a mere theoretical device to characterize the semantics of this output. The value of any "normal" variable which does not occur in a program part is complete given by an equation $v = val$. Variables which appear in program parts are only given implicitly and the set of possible values may actually include many more or less strange things not covered by subsumption.† But it is interesting to note, that although a program which was sought by some query cannot be printed out, it can be executed. If v_p refers to such a program encoded in a success state, we can continue from that state giving queries of the form "... in v_p".

In the end of the paper, we present an example of a derivation process which continues 4.2.

5.5 Correctness of SLG-resolution

We use substitutions to measure and to compare the semantic contents of an initial query and the corresponding success constraints. Our task in this subsection is to show the right relation between the two. So we define the traditional notions . . .

Definition 5.16. A *correct answer substitution* for a query Q is a substitution σ such that for any ground substitution, γ with $Q\theta\gamma$ ground, $Q\theta\gamma$ is true.

A *computed answer substitution* for Q is a unifier for a success constraint for Q.

□

We define the following notion of a solution for an arbitrary resolution state. It is useful when doing proofs which involve intermediate states between the initial and success ones.

Definition 5.17. Let $S = C \cup \{\text{QUERY} = Q\}$ be a resolution state. A substitution is called a *solution* for S, if it is a correct answer substitution for Q and a unifier for C. The notation ss(C) will refer the set of all solutions for C.

□

Proposition 5.9. A solution for an initial resolution state, $\{\text{QUERY} = Q\}$ is the same as a correct answer substitution for Q.

A solution for a success constraint, C, for a query Q, is the same as a unifier for C, i.e., a computed answer substitution for Q.

□

Proof. Follows immediately from the definitions.

□

The following two propositions relate the resolution method's selection procedures for meta-goals and clauses to the corresponding notions used in the definition of the immediate consequence operator, section 4, which refers only to ground instances. In our notation, we consequently use a tilde to indicate ground versions and in the subsequent applications of the propositions, the substitutions named "σ" will typically be solutions.

† This should really not come as a surprise, since there are many, both sensible and seemingly weird, programs which can make a given query true! It is possible — although not very useful — to generalize the construction in the proof of lemma 5.1 to print out an infinite sequence of unifiers which, qua renaming of special constants, is complete in the sense that any unifier can be written as $\sigma\varphi$ for some σ in the sequence. See also example 5.3.

Proposition 5.10. Assume for a meta-goal, M, and query, Q, that

$$\text{select}(M, Q, Q', v_R, E)$$

holds and let σ be a unifier for E. For any ground substitution, γ, such that $Q\sigma\gamma$ and $Q'\{v_R \rightarrow M'\}\sigma\gamma$ are ground queries for some term M', it holds that

$$Q\sigma\gamma = (\widetilde{M_1}, M\sigma\gamma, \widetilde{M_2}), \text{ and}$$

$$Q'\{v_R \rightarrow M'\}\sigma\gamma = (\widetilde{M_1}, M'\sigma\gamma, \widetilde{M_2})$$

for some $\widetilde{M_1}$ and $\widetilde{M_2}$.

The other way round, let Q be a query, σ and γ substitutions, $\overline{M}$ a meta-goal and $\tilde{M}$ a ground meta-goal such that

$$Q\sigma = \cdots, \overline{M}, \cdots \quad \text{and} \quad \overline{M}\gamma = \tilde{M}.$$

Then there exists a meta-goal M, a query Q', a variable v_R which does not occur in Q, and set of equations, E, such that

$$\text{select}(M, Q, Q', v_R, E)$$

holds. Furthermore, there is an extension of σ, $\sigma\varphi$, such that

$$M\sigma\varphi\gamma = \tilde{M}, \text{ and}$$

$\sigma\varphi$ is a unifier for E.

□

Proof. We have to prove, whenever $\text{select}(M, Q, Q', v_R, E)$ holds and σ is a unifier for E, that, for all relevant γ, that the following two equations hold with suitable $\widetilde{M_1}$ and $\widetilde{M_2}$.

$$Q\sigma\gamma = (\widetilde{M_1}, M\sigma\gamma, \widetilde{M_2})$$

$$Q'\{v_R \rightarrow M'\}\sigma\gamma = (\widetilde{M_1}, M'\sigma\gamma, \widetilde{M_2})$$

This is shown by induction over the number of applications of the inductive definition 5.12 which are needed in order to justify that a given select statement holds. We consider each case of definition 5.12 in turns; the first and fifth cases represent the initial induction steps.

Case 1:
Assume $\text{select}(M, M, v_R, v_R, \{\ \})$. In this case, we know nothing about σ and the equations become

$$M\sigma\gamma = M\sigma\gamma, \text{ and}$$

$$v_R\{v_R \rightarrow M'\}\sigma\gamma = M'\sigma\gamma,$$

which corresponds to $\widetilde{M_1}$ and $\widetilde{M_2}$ being empty.

Case 2:
Assume $\text{select}(M, (A, B), (A', B), v_R, E)$ and that the proposition holds for the entities in the statement $\text{select}(M, A, A', v_R, E)$, i.e., for any unifier, σ, for E and suitable γ, the following holds for some $\widetilde{M_1}$ and $\widetilde{M_2}$.

$$A\sigma\gamma = \widetilde{M_1}, M\sigma\gamma, \widetilde{M_2}$$

$$A'\{v_R \rightarrow M'\}\sigma\gamma = \widetilde{M_1}, M'\sigma\gamma, \widetilde{M_2}$$

The corresponding equations for the larger queries follows immediately.

Case 3:
Analogous to case 2.

Case 4:
Assume select$(M, v, Q', v_R, \{v = (v_A, v_B)\} \cup E)$ and that the proposition holds for the entities in the statement select$(M, (v_A, v_B), Q', v_R, E)$, v being a variable, v_A and v_B new variables, i.e., for any unifier, σ, for E and suitable γ, the following holds for some $\widetilde{M_1}$ and $\widetilde{M_2}$.

$$(v_A, v_B)\sigma\gamma = \widetilde{M_1}, M\sigma\gamma, \widetilde{M_2}$$

$$Q'\{v_R \rightarrow M'\}\sigma\gamma = \widetilde{M_1}, M'\sigma\gamma, \widetilde{M_2}$$

About a unifier, σ', for $\{v = (v_A, v_B)\} \cup E$ we know that it is especially a unifier for E and that $v\sigma' = (v_A, v_B)\sigma'$. The equations for the queries v and Q' follows immediately from these properties.

Case 5:
Assume select$(g \text{ in} p, v, v_R, v_R, \{v = (v_g \text{ in } v_p)\})$, v being a variable, v_R, v_g and v_p new variables. Let σ be a unifier for $\{v = (v_g \text{ in } v_p)\}$ and γ as usual. The equation become as follows,

$$v\sigma\gamma = (v_g \text{ in } v_p)\sigma\gamma, \text{ and}$$

$$v_R\{v_R \rightarrow M'\}\sigma\gamma = M'\sigma\gamma,$$

corresponding to $\widetilde{M_1}$ and $\widetilde{M_2}$ being empty.

Case 6:
Assume select$(M, d, Q', v_R, \{v = d\} \cup E)$, d a delayed expression, v a new variable, and that the proposition holds for the entities in the statement select(M, v, Q', v_R, E), i.e., for any unifier, σ, for E and suitable γ, the following holds for some $\widetilde{M_1}$ and $\widetilde{M_2}$.

$$v\sigma\gamma = (\widetilde{M_1}, M\sigma\gamma, \widetilde{M_2})$$

$$Q'\{v_R \rightarrow M'\}\sigma\gamma = (\widetilde{M_1}, M'\sigma\gamma, \widetilde{M_2})$$

The corresponding equation for d follows from the fact that a unifier, σ', for $\{v = d\} \cup E$ especially is a unifier for E and satisfies $v\sigma' = d\sigma'$.

The proof for the second part goes as follows. Assume Q is a query and σ a substitution such that $Q\sigma$ is a query of the form $\cdots, \overline{M}, \cdots$. Assume also a γ such that $\overline{M}\gamma = \widetilde{M}$ is ground. We have to show that there exist M, Q', v_R, and E such that

$$\text{select}(M, Q, Q', v_R, E)$$

and that there is an extension of σ, $\sigma\varphi$, such that

$$M\sigma\varphi\gamma = \widetilde{M}, \text{ and}$$

$\sigma\varphi$ is a unifier for E.

We will prove this by structural induction over $Q\sigma$. Note, that it is very important, when we refer to new variables below, that they are chosen different from those of vars(σ).

Assume $Q\sigma = \overline{M}$. This can be the case if either Q is a variable v, a meta-goal $(g \text{ in } p)$, or a delayed expression d.

- Let $Q = v$ and hence $\sigma(v) = \overline{M}$ and $v\sigma\gamma = \widetilde{M}$. Definition 5.12, case 5 yields

  $$\text{select}(v_g \text{ in } v_p, v, v_R, v_R, \{v = (v_g \text{ in } v_p)\}),$$

where v, v_R, v_g, and v_p are distinct variables, v_R, v_g, and v_p furthermore new variables. An extension $\sigma\varphi$ of σ which assigns the immediate subterms of $\overline{M}$ to v_g and v_p is a unifier for the equation $v = (v_g \text{ in } v_p)$ and $(v_g \text{ in } v_p)\sigma\varphi\gamma = \widetilde{M}$.

- Let $Q = (g \text{ in } p)$ for some terms g and p. Definition 5.12, case 1 yields

  $$\text{select}(Q, Q, v_R, v_R, \{\,\}),$$

  where v_R is variable which does not occur in Q. The substitution σ is unifier for the empty set of equations and $Q\sigma\gamma = \widetilde{M}$.

- Let $Q = d$ for some delayed expression d and hence $d\sigma = \overline{M}$ and $d\sigma\gamma = \widetilde{M}$. Definition 5.12, cases 5 and 6 combined, yields

  $$\text{select}(v_g \text{ in } v_p, d, v_R, v_R, \{v = d, v = (v_g \text{ in } v_p)\}),$$

  where v, v_g, and v_p are new variables. The relevant extension of σ assigns $\overline{M}$ to v and the immediate substructures of $\overline{M}$ to v_g and v_p.

Assume $Q\sigma = (\overline{A}, \overline{B})$ and $\overline{A}\gamma = \widetilde{A}$, $\overline{B}\gamma = \widetilde{B}$. Assume also that $\widetilde{A} = \cdots, \widetilde{M}, \cdots$ (the case where $\widetilde{M}$ is part of $\widetilde{B}$ is shown in a similar way). This is possible in three ways,

a) $Q = v$, v a variable, in which case $\sigma(v) = (\overline{A}, \overline{B})$, or
b) $Q = (A, B)$, with $A\sigma = \overline{A}$, $B\sigma = \overline{B}$ or
c) $Q = d$, d a delayed expression with $d\sigma = (\overline{A}, \overline{B})$.

We consider case a). Let v_A and v_B be new variables and φ_1 be the substitution $\{v_A \to \overline{A}, v_B \to \overline{B}\}$. By induction we know that there exists M, A', v_R, and E such that

$$\text{select}(M, v_A, A', v_R, E)$$

and an extension of $\sigma\varphi_1$, $\sigma\varphi_1\varphi_2$, which is a unifier for E and $M\sigma\varphi_1\varphi_2\gamma = \widetilde{M}$. Definition 5.12, case 2, gives now, that

$$\text{select}(M, (v_A, v_B), (A', v_B), v_R, E)$$

and definition 5.12, case 4, in turn that

$$\text{select}(M, v, (A', v_B), v_R, \{v = (v_A, v_B)\} \cup E)$$

The substitution $\sigma\varphi_1\varphi_2$ is clearly a unifier for $\{v = (v_A, v_B)\} \cup E$.

In cases b and c, we get the conclusion in a similar way using definition 5.12, cases 2 and 6+2+4, respectively.

□

Proposition 5.11. Assume for a clause, c, and generalized term, t, that

$$\text{member}(c, t, E)$$

holds and let σ be a unifier for E. Then it holds that

$$t\sigma = [\cdots, c\sigma, \cdots].$$

The other way round, let t be a generalized term and σ and γ substitutions, $\overline{c}$ a clause, $\widetilde{c}$ a ground clause such that

$$t\sigma = [\cdots, \overline{c}, \cdots] \quad \text{and} \quad \overline{c}\gamma = \widetilde{c}.$$

Then there exists a clause c and equations E such that

$$\text{member}(c, t, E)$$

holds. Furthermore, there is an extension of σ, $\sigma\varphi$, such that

$c\sigma\varphi\gamma = \tilde{c}$, and

$\sigma\varphi$ is a unifier for E.

□

Proof. We will prove that, whenever member(c, t, E) holds and σ is a unifier for E, that $t\sigma$ has the form $[\cdots, c\sigma, \cdots]$. This is shown by induction over the number of applications of the inductive definition 5.13 which are needed in order to justify that a given member statement holds. We consider each case of definition 5.13 in turns; the first case represents the initial induction step.

Case 1:
Assume member$((H :- B), [(H' :- B')|t], E)$ and let σ be a unifier for the (possibly empty) set of equations, E. We see that either H is identical H' or E contains the equation $H = H'$ so in any case $H\sigma = H'\sigma$, and in a similar way we get $B\sigma = B'\sigma$. So we conclude that $[(H' :- B')|t]\sigma = [(H :- B)|t]\sigma = [(H\sigma :- B\sigma)|t\sigma]$ (i.e., referring to the notation above, the first ellipsis is empty, the second is $t\sigma$.

Case 2:
Assume member$(c, [v|t], \{v = (v_h :- v_b)\} \cup E)$, v_h and v_b new variables, and that the proposition holds for member$(c, [(v_h :- v_b)], E)$, i.e., for any unifier, σ, for E, that

$$c\sigma = (v_h :- v_b)\sigma.$$

About a unifier, σ', for $\{v = (v_h :- v_b)\} \cup E$ we know that it is especially a unifier for E and that $v\sigma' = (v_h :- v_b)\sigma'$. The conclusion follows directly by insertion.

The remaining cases 3–6 goes in exactly the same manner.

The proof of the second part goes as follows. Assume t is a generalized term and σ a substitution such that $t\sigma$ is a list of the form $[\cdots, \bar{c}, \cdots]$, where $\bar{c}\gamma = \tilde{c}$ is a ground clause. We have to show that there exist c and E such that

member(c, t, E)

and an extension of σ, $\sigma\varphi$, such that

$c\sigma\varphi\gamma = \tilde{c}$, and

$\sigma\varphi$ is a unifier for E.

We will prove this by structural induction over $t\sigma$ and, in each case, over the structure of t according to the following scheme. With respect to the structure of t we distinguish between a) $t\sigma = [\bar{c}, \cdots]$, and b) $t\sigma = [\cdots, \bar{c}, \cdots]$ (including $t\sigma = [\cdots, \bar{c}]$). We subdivide according to the structure of t as follows; it is understood that v always is a variable and that $\bar{c}$ occurs in $t\sigma$ in a sub-structure corresponding to the emphasized sub-structure of t. For each case we indicate which clause of definition 5.13 is used to reduce it into which other cases.

a-1) $t = v$, reduced into one of the cases a-2 or a-3 using clause 5.

a-2) $t = [v, \cdots]$, reduced into case a-3 using clause 2.

a-3) $t = [(H' :- B'), \cdots]$, base case, proved using clause 1, considered in detail below.

a-4) $t = \mathcal{R}_i\mathcal{D}[\![t']\!]$, reduced into case a-1 using clause 6.

a-5) $t = [\mathcal{R}_i\mathcal{D}[\![t']\!], \cdots]$, reduced into case a-2 using clause 3.

b-1) $t = v$, reduced into one of the cases b-2 or b-2′ using clause 5.

b-2) $t = [\cdots, v, \cdots]$, reduced into case a-2 using clause 4 a number of times.

b-2′) $t = [\cdots | v]$, reduced into case a-1 using clause 4 a number of times.

b-3) $t = [\cdots, (H' :- B'), \cdots]$, reduced into case a-3 using clause 4 a number of times.

b-4) $t = \mathcal{R}_i\mathcal{D}[\![t']\!]$, reduced into case b-1 using clause 6.

b-5) $t = [\cdots, \mathcal{R}_i\mathcal{D}[\![t']\!], \cdots]$, reduced into case a-5 using clause 4 a number of times.

b-5′) $t = [\cdots | \mathcal{R}_i\mathcal{D}[\![t']\!]]$, reduced into case b-4 using clause 4 a number of times.

To ensure that this actually is an induction proof, we have to justify that each case eventually is reduced into the base case a-3. We see this as follows; let > stand for "reduced into". For the b-cases we have that

$$\text{b-5}' > \text{b-4} > \text{b-1} > \{\text{b-2}, \text{b-2}'\}$$

All b-cases reduce in a-cases:

$$\text{b-2} > \text{a-2}, \quad \text{b-2}' > \text{a-1}, \quad \text{b-3} > \text{a-3}, \quad \text{b-5} > \text{a-5}.$$

For the a-cases we have

$$\text{a-4} > \text{a-1} > \{\text{a-2}, \text{a-3}\} \text{ and } \text{a-5} > \text{a-2} > \text{a-3}.$$

We will only consider the base case a-3 in detail; the other case are trivial and go as in the proof of proposition 5.10.

So assume $t\sigma = [\bar{c}, \cdots]$ where $\bar{c} = (\bar{g} \text{ in } \bar{p} :- \bar{b})$ and that $t = [(H' :- B'), \cdots]$. According to clause 1 of definition 5.13, there exist H, B, and E such that

$$\text{member}((H :- B), t, E)$$

where H, B, and E depend on H' and B'. We analyze the possibilities and construct a substitution φ accordingly.

- If H' is a variable v_h, then $H = (v_g \text{ in } v_p)$ where v_g, v_p are new variables and E will contain the equation $v_h = (v_g \text{ in } v_g)$.
  In this case it must hold that $\sigma(v_h) = (\bar{g} \text{ in } \bar{p})$ and we let $\varphi(v_g) = \bar{g}$ and $\varphi(v_p) = \bar{p}$.
- If H' is not a variable but a meta-goal, we have $H = H'$ and that E contains no contribution for the head of the clause and we add nothing to φ.
- If B' is a variable v_b,
  - if $\sigma(v_b) = \text{TRUE}$, we choose $B = \text{TRUE}$ and $(v_b = \text{TRUE}) \in E$,
  - otherwise, choose $B = v_b$ in which case E does not contain any equation concerned with the body.
- If B' is a structure, E does not contain any equation concerned with the body.
- If B' is a delayed expression d,
  - if $d\sigma = \text{TRUE}$, we choose $B = \text{TRUE}$ and $(\text{TRUE} = d) \in E$,
  - otherwise, choose $B = d$ in which case E does not contain any equation concerned with the body.

The φ thus constructed is clearly such that $\sigma\varphi$ is an extension (as opposed to a specialization) of σ since φ only touches new variables and $\sigma\varphi$ is a unifier for E and $(H :- B)\sigma\varphi\gamma = \tilde{c}$.
□

We combine the two previous propositions into a statement about resolution steps.

Proposition 5.12. Assume that $S = C \cup \{\text{QUERY} = Q\}$ and $S' = C' \cup \{\text{QUERY} = Q'\}$ are resolution states such that S' is derived from S with selected goal M and clause $H :- B$. If σ is a unifier for C' and γ is such that $Q\sigma\gamma$ and $Q'\sigma\gamma$ are ground, then the following holds for some $\widetilde{M_1}$ and $\widetilde{M_2}$.

$$Q\sigma\gamma = \widetilde{M_1}, M\sigma\gamma, \widetilde{M_2} \text{ and}$$
$$Q'\sigma\gamma = \widetilde{M_1}, B\sigma\gamma, \widetilde{M_2}$$

where $M\sigma\gamma = H\sigma\gamma$ and $(H :- B)\sigma\gamma$ is self-contained.

The other way round, let $S = C \cup \{\text{QUERY} = Q\}$ be a resolution state and σ a unifier for C and γ a substitution such that

$$Q\sigma = \cdots, \overline{M}, \cdots \quad \text{and} \quad \overline{M}\gamma = \widetilde{M}$$

where $\widetilde{M}$ is a ground meta-goal of the form $(\widetilde{g}\ \text{in}\ \widetilde{p})$. Assume that $\mathcal{D}[\![\widetilde{p}]\!]$ has an instance of the form

$$[\cdots, (\widetilde{M} :- \widetilde{B}), \cdots].$$

Then there exists a derivation from S to another state, S', and an extension of σ, $\sigma\varphi$, which is a unifier for the equations created in the selection and derivation steps. If, furthermore, σ is a solution for S, φ can be chosen such that $\sigma\varphi$ is a solution to S'.
□

Proof. For the first part of the proposition, assume the notation used in the proposition and, furthermore, that the selection steps involves replacement position v_R and additional equations E. That σ is a unifier for C' implies that σ is a unifier for the equations $E \cup \{M = H, v_R = B\}$, cf. lemma 5.2 (correctness of the normalization algorithm). Proposition 5.10 gives that

$$Q\sigma\gamma = \widetilde{M_1}, M\sigma\gamma, \widetilde{M_2} \text{ and}$$
$$\hat{Q}\sigma\gamma = \widetilde{M_1}, B\sigma\gamma, \widetilde{M_2}$$

for some $\widetilde{M_1}$ and $\widetilde{M_2}$ where $\hat{Q}$ is the intermediate query which results from the selection of M from Q. The desired equation for Q' follows from the fact that the sort changes the normalization algorithm can do when changing $\hat{Q}$ into Q' is to exchange terms s for terms t such that $\sigma\gamma$ is a unifier for the equation $s = t$.

That $M\sigma\gamma = H\sigma\gamma$ follows from σ being a unifier for the equation $M = H$.

The self-containedness of $(H :- B)\sigma\gamma$ is seen as follows. Assume $M = (g_M\ \text{in}\ p_M)$ and $H = (g_H \text{in} p_H)$. From the equation above, we get that $p_M\sigma = p_H\sigma$. The definition of derivation step gives that

$$\text{member}((H :- B), \mathcal{R}_i\mathcal{D}[\![p_M]\!], E)$$

holds for some equations E for which σ (also) is a unifier and proposition 5.11 gives the following.

$$\mathcal{R}_i\mathcal{D}[\![p_M]\!]\sigma = [\cdots, (H :- B)\sigma, \cdots]$$

Apply, now, γ to either side, the rule for distribution of substitutions over $\mathcal{R}_i\mathcal{D}[\![-]\!]$ and $p_M\sigma = p_H\sigma$ and we get

$$\mathcal{R}_i\mathcal{D}[\![p_H\sigma\gamma]\!]\sigma\gamma = [\cdots, (H :- B)\sigma\gamma, \cdots]$$

This shows that the ground clause instance $(H :- B)\sigma\gamma = (g_H\sigma\gamma\ \text{in}\ p_H\sigma\gamma :- B\sigma\gamma)$ is self-contained.

The proof of the second part goes as follows. Assume a state $S = C \cup \{\text{QUERY} = Q\}$, a unifier, σ for C and a substitution γ such that

$$Q\sigma = \cdots, \overline{M}, \cdots \quad \text{and} \quad \overline{M}\gamma = \widetilde{M} \text{ where}$$

$$\widetilde{M} = \widetilde{g} \text{ in } \widetilde{p}.$$

Assume also that $\mathcal{D}[\![\widetilde{p}]\!]$ has an instance

$$[\cdots, (\widetilde{M} :- \widetilde{B}), \cdots].$$

Proposition 5.10 gives M, Q', v_R, and E_1 such that

$$\text{select}(M, Q, Q', v_R, E_1)$$

and a φ_1 such that $M\sigma\varphi_1\gamma = \widetilde{M}$ and $\sigma\varphi_1$ is a unifier for E_1. Let $M = (g \text{ in } p)$ and proposition 5.11 (applied with the substitution $\sigma\varphi_1$) gives a clause $(H :- B)$ and E_2 such that

$$\text{member}((H :- B), \mathcal{D}[\![p]\!], E_2)$$

and a φ_2 such that $\sigma\varphi_1\varphi_2$ is a unifier for E_2 and $(H :- B)\sigma\varphi_1\varphi_2\gamma = (\widetilde{M} :- \widetilde{B})$.

If we now let $\varphi_3(v_R) = B\sigma\varphi_1\varphi_2$ and $\varphi_3(\text{QUERY}) = Q'\{v_R \rightarrow B\}\sigma\varphi_1\varphi_2$ we see that $\theta\varphi_1\varphi_2\varphi_3$ is a unifier for

$$C \cup E_1 \cup E_2 \cup \{M = H, v_R = B\} \cup \{\text{QUERY} = Q'\}.$$

Hence the normalization algorithm, when applied to this constraint, will return a state $S' = C' \cup \{\text{QUERY} = Q''\}$, i.e., S' is derived from S. Lemma 5.2 tells that, if we let $\varphi = \varphi_1\varphi_2\varphi_3$, $\sigma\varphi$ is a unifier for S'. Finally, assume additionally that σ is a solution for S. We saw above that $\sigma\varphi$ is a unifier for $C' \subseteq S'$ and $Q''\sigma\varphi\gamma$ is a true query since $Q'\sigma\varphi = Q\sigma\varphi$. The last conclusion is a consequence of the fact that the normalization algorithm only exchanges subterms of Q' when turning it into Q'' which are indifferent with respect to $\sigma\varphi$. Hence $\sigma\varphi$ is a solution for S'.

□

The following lemma gives the exact relation between the solutions for a given state and the states derived from it.

Lemma 5.4. Let S be a resolution state which is not a success state and let $\{S_i\}_{i \in I}$ be the set of all resolution states which can be derived in a single step from S.

Then $\text{ss}(S)$ is a restriction of $\bigcup_{i \in I} \text{ss}(S_i)$.

□

Proof. We will prove that any solution for some S_i also is a solution for $S = C \cup \{\text{QUERY} = Q\}$ and that any solution for S can be extended to a solution for some S_i.

So let σ_i be a unifier for $S_i = C_i \cup \{\text{QUERY} = Q_i\}$. The constraint C_i is constructed from C by adding some new equations and applying the normalization algorithm. According to lemma 5.2, σ_i is also a unifier for C. That σ_i is a correct answer substitution for Q follows from the first half of proposition 5.12.

The other way round, let σ be a solution for S and γ a substitution such that $S\sigma\gamma$ is ground. Since S is not a success state but S has solutions, it must hold that

$$S\sigma\gamma = \cdots, \widetilde{M}, \cdots \text{ with } \widetilde{M} \in \mathcal{M}^{\text{GCP}}.$$

By definition of a model, the conditions in the second half of 5.12 are satisfied, and this gives an S_i derived from S and a φ such that $\sigma\varphi$ is a solution for S_i.

□

Theorem 5.1. Soundness and completeness of SLG-resolution.
Let Q be a query and $\{S_i\}_{i\in I}$ the set of all possible success states in resolutions of Q.
Then ss($\{$QUERY $= Q\}$) is a restriction of $\bigcup_{i\in I}$ ss(S_i).
□

Proof. By induction using lemma 5.4 we get the following property:

Let S be an initial state and $SS_n = \{S_i\}_{i\in I}$ the set of all success states derived in less that n steps together with all states which can be derived in exactly n steps from S. Then ss(S) is a restriction of $\bigcup_{i\in I}$ ss(S_i).

The theorem follows now from the fact that $\lim_{n\to\infty} SS_n$ is the set of all success states.
□

Soundness and completeness are formulated more traditionally in the following corollaries which are direct consequences of the theorem.

Corollary 5.1. Soundness.
Let σ' be a computed answer substitution for Q. Then σ' is a correct answer substitution for Q.
□

Proof. Assume S_i is a success state in a resolution starting with an initial state, $S = \{$QUERY $= Q$. A solution, σ_i, for S_i can be written as $\sigma\varphi$ with $\sigma \in$ ss(S); hence any such σ_i is a correct answer substitution for Q.
□

Corollary 5.2. Completeness.
Let σ be a correct answer substitution for Q. Then there exists a computed answer substitution, σ' such that $\sigma\varphi = \sigma'$ for some φ.
□

Proof. Let σ be a solution for an initial state S. Then the theorem states that it can be found as an extension in some ss(S_i).
□

5.6 Weak resolution, the Bowen-Kowalski interpreter as a special case

The reason why we had to introduce the complicated matters above was to take action in the case where a function $\mathcal{R}_i\mathcal{D}[\![-]\!]$ is applied to a non-ground term. We can formulate an alternative resolution method — called weak SLG-resolution — which only generates paths of derivation steps in which program parts are ground. Hence any $\mathcal{R}_i\mathcal{D}[\![-]\!]$ function symbol is reduced away immediately before it can sneak into the resolution state or wake up the oracles. If a non-ground program part is reached by a step, an implementation could issue an error message or maybe delay that particular meta-goal until the problematic variables perhaps get grounded. This is obviously not complete but many meta-programming problems are such that the condition can be obeyed. Actually, we arrive (not surprisingly) at a resolution method which resembles Bowen and Kowalski's (1982) sketch of an interpreter for a demo(–, –) predicate which we criticized in the introduction.

Definition 5.18. A *weak* (resolution) state is a state

$$S = C \cup \{\text{QUERY} = Q\}$$

where S does not contain delayed expressions and Q is a sequence of meta-goals.

A weak state, S', is *derived weakly* from another weak state, $S = C \cup \{\text{QUERY} = Q\}$, if the following holds.

- Assume $Q = M_1, \cdots, M_i, \cdots, M_n$ such that

$$M_i = G \text{ in } P,$$

- if $\mathcal{R}_i\mathcal{D}[\![P]\!]$ is not a program, there is no such S', otherwise let

$$\mathcal{R}_i\mathcal{D}[\![P]\!] = [\cdots, (H :\!- B), \cdots],$$

- S' is a state returned by the normalization algorithm when applied to

$$C \cup \{M_i = H\} \cup \{\text{QUERY} = M_1, \cdots, M_{i-1}, B, M_{i+1}, \cdots, M_n\}.$$

□

We observe here that we need only the first four cases in the normalization algorithm, i.e., those that refer to usual terms only. — And we end up with exactly the unification algorithm given by Martelli and Montanari (1982) for plain logical programs.

The correctness of weak SLG-resolution, i.e., soundness, can be formulated as follows.

Theorem 5.2. Soundness of weak SLG-resolution.
Let Q be a query and $\{S_i\}_{i\in I}$ the set of all possible success states in weak resolutions of Q.

Then $\text{ss}(\{\text{QUERY} = Q\})$ has a restriction of $\bigcup_{i\in I} \text{ss}(S_i)$ as a subset.

□

Proof. Follows from theorem 5.1 since weak SLG-resolution clearly is a specialized version of general SLG-resolution.

□

This compact theorem implies soundness as in the general case, any weakly computed answer substitution is a correct answer substitution. We may furthermore say that weak SLG-resolution is "weakly complete" in the sense that it (by definition) can find any correct solution which can be reached by proofs which involves only selections of clauses from ground program parts.

So viewing weak SLG-resolution as an interpretation algorithm, our theorem is slightly more general that the correctness theorems for meta-interpreters considered by Hill and Lloyd (1989) in that weak SLG-resolution also covers (using their terminology)

- calls of the meta-interpreter from meta-interpreted code (by our choice of language), and
- programs which generate new programs dynamically — with the restriction, that these programs are fully generated when they are reached by the interpreter.

Hill and Lloyd represent the program being executed by the interpreter as global facts, i.e., they must by fully given from the beginning.

6 Conclusion and perspectives

We have shown how the language of plain definite clauses, or pure Prolog, can be generalized into a meta-programming language with reflexive facilities corresponding to a binary demo predicate. A generalization which preserves the following characteristics, for which logic programming usually is appreciated,

- a declarative semantics based on relatively simple, model-theoretical concepts, and
- a resolution method which is sound and complete with respect to the declarative semantics.

To our knowledge, the importance of these central qualities has not been recognized properly in earlier work on such languages.

Completeness of resolution implies an until now overlooked potentiality for a more declarative style of meta-programming, especially for program generation. A meta-program can state that this-and-this should be provable in a desired program together with certain syntactic restrictions — and the interpreter will do the program synthesis. This as opposed to current technology in which a meta-program for program generation typically appears as a detailed recipe for program synthesis. It should be made clear, however, when we outline these perspectives, that what we have done in the present work is concerned solely with the linguistic foundations, the practical evidence has yet to be given.

The particular kind of ground representation of programs as data was chosen as simple as possible. Variables are exchanged with special constants, otherwise programs and representations are isomorphic. Consequently, we have defined programs (and their representations) to be lists, as opposed to sets, of clauses. This implies a strictly syntactic notion of program equivalence, so if an interpreter prints out a program as a solution, it may also — actually it should — be able to print out all other programs which arise from a permutation of the clauses in the program and meta-goals in clause bodies. Two programs are equivalent, i.e., their representations unify, if and only if they possess exactly the same structure, even down to variable names. In the discussion of the selection procedure for clauses, definition 5.13, — which also serves as an oracle for program and clause creation when needed — we considered the relevance of extending the resolution to work explicitly with constraints such as "L is a list with X as an element". This may be an opening for treating programs as sets of clauses and similarly for the meta-goals in the body of a clause. With respect to equivalence of variables by their name, it is interesting to note that our resolution method never needs to generate new special constants which serve as variable names.

Future directions

For an implementation of SLG-resolution, we need a more efficient representation than the systems of equations we have described with the implied extensive pattern matching needed in the normalization algorithm. We need a more efficient kind of a state with some sort of structure sharing, e.g., (Boyer, Moore, 1972, Mellish†, 1982). However, due to the functions in the state, we need some explicit change propagation in addition to the instant propagation provided by structure sharing; such hybrid systems of dependencies has been studied in (Christiansen, 1989b). Another thing that would be interesting to know more about is whether we can drop the occur check (cf. step 2 in our normalization algorithm) as is custom in Prolog implementations for reasons of efficiency. Programming in Prolog works quit well in practice even with the risks implied by the missing occur check. Will this be the same for SLG-resolution or is there too much potential circularity inherent in the concept? Good question.

It would also be interesting to know more about negation in this context, both with respect to the declarative and procedural aspect. With the present methods we can ask for a program in which certain statements are true; it will also be interesting to ask for programs in which some statements are known to be false.

This leads us to the question about methodology. Our work can be seen as a step towards the development of an automaton or a programming robot which we can order about as follows, "Give me a program with such-and-such properties", and the robot will prepare a program, perhaps not a program which satisfies our needs but at least one that meets our

† Actually, Mellish refers to his approach as structure copying, but it really means structure sharing *á la* Lisp.

specification. We have to learn to ask this sort of questions such that we are able to inquire, not only "Give me a sorting program," but also "Give me a *good* sorting program" . . .

A final example

We will illustrate some of the points above be an example.

Example 6.1. We will continue example 4.2, so let $\mathcal{P}$ stand for the following non-ground term.

$$[(p(*y) \text{ in } *x :- r \text{ in } *x, Z), (r \text{ in } *x :- \textsc{True}), (q(a) \text{ in } *x :- \textsc{True})]$$

We will consider a resolution process with the following initial query.

$$p(X) \text{ in } \mathcal{P}$$

In the first resolution step, there is only one possible meta-goal to select, and a clause will be selected from the value of $\mathcal{R}_1\mathcal{D}[\![\mathcal{P}]\!]$. Assume the first clause is selected, i.e.,

$$p(y_0) \text{ in } x_0 :- r \text{ in } x_0, \mathcal{R}_0\mathcal{D}[\![Z]\!],$$

assuming $\mathcal{R}_0\mathcal{D}[\![*y]\!]$ yields the variable y_0 and so on. We get the following constraint before normalization is applied; v_R^0 is the replacement variable.

$$\begin{aligned}&\{(p(X) \text{ in } \mathcal{P}) = (p(y_0) \text{ in } x_0),\\ &\ v_R^0 = (r \text{ in } x_0, \mathcal{R}_0\mathcal{D}[\![Z]\!])\}\\ &\cup\{\textsc{Query} = v_R^0\}\end{aligned}$$

Applying the normalization algorithm, we get the following, derived state.

$$\begin{aligned}&\{X = y_0,\\ &\ x_0 = \mathcal{P},\\ &\ v_R^0 = \cdots\}\\ &\cup\{\textsc{Query} = (r \text{ in } \mathcal{P}, \mathcal{R}_0\mathcal{D}[\![Z]\!])\}\end{aligned}$$

Let's take a derivation step at level 1. We let the selection procedure for meta-goals choose the first meta-goal in the current query, $(r \text{ in } \mathcal{P}$, and we get an intermediate query $(v_R^1, \mathcal{R}_0\mathcal{D}[\![Z]\!])$. We apply $\mathcal{R}_1\mathcal{D}[\![-]\!]$ to the program part of the selected meta-goal and let the member procedure choose the second clause, i.e.,

$$r \text{ in } x_1 :- \textsc{True}.$$

We put together the relevant equations and apply the normalization algorithm, and we get the following derived state.

$$\begin{aligned}&\{X = y_0,\\ &\ x_0 = \mathcal{P},\\ &\ x_1 = \mathcal{P},\\ &\ v_R^0 = \cdots,\ v_R^1 = \cdots\}\\ &\cup\{\textsc{Query} = (\textsc{True}, \mathcal{R}_0\mathcal{D}[\![Z]\!])\}\end{aligned}$$

For the next step, the meta-goal has to be selected in $\mathcal{R}_0\mathcal{D}[\![Z]\!]$ and we will assume that the oracle does not feel for expanding it to a sequence of several meta-goals, but only provides the necessary structure for turning $\mathcal{R}_0\mathcal{D}[\![Z]\!]$ into a meta-goal. This creates the following additional equations,

$\{v = \mathcal{R}_0\mathcal{D}[\![Z]\!], v = (v_g \text{ in } v_p)\},$

and $(v_g \text{ in } v_p)$ is the selected goal. We apply $\mathcal{R}_2\mathcal{D}[\![-]\!]$ to v_p, which yields the delayed expression $\mathcal{R}_2\mathcal{D}[\![v_p]\!]$ which in turn is given to the member procedure. Assume, now, that the oracle in the member procedure feels for the unknown program to have three or more clauses and that the selected clause should be the third one and finally, that this clause should be a fact. This yields a terrible amount of additional equations which we add to the current constraint together with the equation which unifies the selected meta-goal with the head the of selected clause, and finally the equation about the replacement variable. We normalize and get the following state; equations about uninteresting variables are suppressed by ellipses.

$\{X = y_0,$
$x_0 = \mathcal{P},$
$\cdots,$
$\mathcal{R}_0\mathcal{D}[\![Z]\!] = (v_g \text{ in } v_p),$
$\mathcal{R}_2\mathcal{D}[\![v_p]\!] = [v_1, v_2, (v_g \text{ in } v_p :\!- \text{ TRUE})|v_t],$
$\cdots,$
$v_R^0 = \cdots,\ v_R^1 = \cdots,\ v_R^2 = \cdots\}$
$\cup\{\text{QUERY} = (\text{TRUE}, \text{TRUE})\}$

This state is a success state and according to lemma 5.1, it is guaranteed to have solutions, i.e., computed answers for the initial query. The proof of the lemma is constructive, it simply assigns suitable special constants as the values of variables in delayed expressions and adjusts the value of all other variables accordingly. Let us apply this construction to the constraint above. We put in $*a$ for Z and $*b$ for v_p and the critical equations become as follows.

$\mathcal{R}_0\mathcal{D}[\![*a]\!] = (v_g \text{ in } *b)$
$\mathcal{R}_2\mathcal{D}[\![*b]\!] = [v_1, v_2, (v_g \text{ in } *b :\!- \text{ TRUE})|v_t]$

So, if $\mathcal{R}_0\mathcal{D}[\![*a]\!]$ and $\mathcal{R}_2\mathcal{D}[\![*b]\!]$ yield the respective variables A_0 and B_2, the construction gives the following computed answer.

$\{Z \rightarrow *a, v_p \rightarrow *b, A_0 \rightarrow (v_g \text{ in } *b), B_0 \rightarrow [v_1, v_2, (v_g \text{ in } *b :\!- \text{ TRUE})|v_t], \cdots\}$

So the construction in the proof is only a rough proof of existence. The constructed substitution's statement about the variable Z can, with a little irony, be phrased as "The query will succeed whenever Z denotes a variable because a variable always unifies with any trivially true meta-goal".

Consider the following substitution, which by definition also is a computed answer for the initial query.

$\{Z \rightarrow (q(*a) \text{ in } *x), X \rightarrow a, y_0 \rightarrow a, v_p \rightarrow \mathcal{P}', \cdots\}$

where $\mathcal{P}'$ is as $\mathcal{P}$ but with Z replaced by $(q(*y) \text{ in } *x)$

From a subjective point of view, we may find this computed answer "better" than the one given by the proof of lemma 5.1 as shown above. Why is it better? Not for any mathematical reason, since mathematics does not recognize properties such as goodness. In an attempt to psychoanalyze ourselves, to see why we are more satisfied with the latter answer, it seems that we prefer solutions which somehow reuses bits and pieces of the original program.

So assume that we for some methodological reason has extended the query with syntactic requirements such that possible values of Z which are synthesized from pieces of $\mathcal{P}$ are tested against the constraint already found. Eventually, the equation

$$Z = (q(*y) \text{ in } *x)$$

will appear. The normalization algorithm will quickly unveil a new success state.

We hope with this example to have illustrated our resolution method and also to have shown the motivation for the future development of a methodology for using it.
□

References

Abramson, H., and Rogers, M.H., eds., *Meta-programming in Logic Programming*. MIT Press, 1989.

Bacha, H., Meta-level programming: A compiled approach. *Logic Programming, Proceedings of the fourth international conference*, ed. Lassez, J.-L., MIT Press, pp. 394–410, 1987.

Bacha, H., MetaProlog design and implementation. *Logic Programming, Proceedings of the fifth international conference*, ed. Kowalski, R.A., Bowen, K.A., MIT Press, pp. 1371–1387, 1988.

Bonnier, S., Horn clause logic with external procedures: Towards a theoretical framework. *Linköping Studies in Science and Technology*, Thesis no. 197. Linköping University, 1989.

Boolos, G., *The Unprovability of Consistency, An essay in modal logic*. Cambridge University Press, 1979.

Bowen, K.A., Meta-level programming and knowledge representation. *New Generation Computing* 3, pp. 359–383, 1985.

Bowen, K.A. and Kowalski, R.A., Amalgamating language and meta-language in logic programming. *Logic Programming*, Clark, K.L. and Tärnlund, S.Å., eds., pp. 153–172, Academic Press, 1982.

Bowen, K.A. and Weinberg, T., A meta-level extension of Prolog. *IEEE Symposium on Logic Programming*, pp. 669–675, Boston, 1985.

Boyer, R.S. and Moore, J.S., The sharing of structure in theorem-proving programs. *Machine Intelligence* 7, pp. 101–106, 1972.

Bratko, I., *Prolog, Programming for artificial intelligence*. Addison-Wesley, 1986.

Brogi, A., Lamma, E., and Mello, P., Hypothetical reasoning in logic programming: A semantic approach. *Information Processing Letters* 36, pp. 285–291, 1990.

Burt, A.D., Hill, P.M., and Lloyd, J.W., *Preliminary Report on the Logic Programming Language Gödel*. Dept. of Computer Science, Univ. of Bristol, TR-90-02, 1990.

Bruynooghe, M., ed., *Proceedings of the Second Workshop on Meta-programming in Logic*. April 4–6, 1990, Leuven, Belgium.

Christiansen, H., Syntax, semantics, and implementation strategies for programming languages with powerful abstraction mechanisms. *Proc. 18th Hawaii International Conference on System Sciences*, vol. 2, pp. 57–66, 1985.

Christiansen, H., Context-sensitive parsing in full Prolog, *Datalogiske skrifter*, no. 5, Roskilde University Centre, 1986a.

Christiansen, H., Recognition of Generative Languages, Lecture Notes in Computer Science 217, pp. 63-81, Springer-Verlag, 1986b.

Christiansen, H., The syntax and semantics of extensible languages. *Datalogiske skrifter*, no. 14, Roskilde University Centre, 1988.

Christiansen, H., Extensible logic for extensible languages, *Datalogiske skrifter*, no. 21, Roskilde University Centre, 1989a.

Christiansen, H., Structure sharing in incremental systems, *Structured Programming*, 10/4, pp. 169–186, 1989b.

Christiansen, H., Declarative semantics of a meta-programming language. *In:* Bruynooghe, 1990, pp. 159–168, 1990a.

Christiansen, H., A survey of adaptable grammars, *SIGPLAN Notices* 25/11, pp. 35–44, 1990b.

Costantini, S., Semantics of a metalogic programming language. *In:* Bruynooghe, 1990, pp. 3–18.

Costantini, S. and Lanzarone, G.A., Towards metalogic programming, in: Martelli, A., Valle, G., (eds.) *Proceedings of Computational Intelligence 88*, pp. 41–52. North-Holland, 1988.

Deransart, P. and Maluszynski, J., Relating logic programs and attribute grammars. *Journal of Logic Programming* 2, pp. 119–155, 1985.

van Emden, M.H. and Kowalski, R.A., The semantics of predicate logic as a programming language. *Journal of the ACM*. Vol. 23, pp. 733–742, 1976.

Enderton, H.B., *A Mathematical Introduction to Logic*. Academic Press, 1972.

Gabbay, D.M. and Reyle, U., N-Prolog: An extension of Prolog with hypothetical implications. *Journal of Logic Programming* 2, pp. 319–355, 1984.

Giordano, L. and Martelli, A., A modal reconstruction of blocks and modules in logic programming. *To appear in the proceedings of International Logic Programming Symposium*, 1991.

Giordano, L., Martelli, A., and Rossi, G., Extending Horn clause logic with implication goals. *To appear in Theoretical Computer Science*, 1991.

Goguen, J.A. and Burstall, R.M., Introducing institutions. *Lecture Notes in Computer Science* 164, pp. 221–256, 1984.

Goguen, J.A. and Mesequer, J., Equality, types, modules and (why not?) generics for logic programming. *Journal of Logic Programming* 2, pp. 179–210, 1984.

Hill, P.M. and Lloyd, J.W., Analysis of meta-programs. *In:* Abramson, Rogers, 1989, pp. 23–51.

Jaffar, J. and Lassez, J.-L., Constraint logic programming. *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pp. 111–119, 1987.

Jaffar, J. and Michaylov, S., Methodology and implementation of a CPL system. *Logic Programming, Proceedings of the fourth international conference*, ed. Lassez, J.-L., MIT Press, pp. 196–218, 1987.

Ko, H.-P. and Nadel, M.E., Substitution and refutation revisited. *Logic Programming, Proceedings of the eighth international conference*, ed. Furukawa, K., MIT Press, pp. 679–692, 1991.

Kowalski, R.A., *Logic for Problem Solving*, Elsevier-North Holland, 1979.

Kripke, S., Semantical analysis of modal logic I. Normal modal propositional calculi. *Zeitschrift für matematische Logik und der Grundlagen der Mathematik* 9, pp. 67–96, 1963

Kripke, S., Outline of a theory of truth, *The Journal of Philosophy* 72, pp. 690–716, 1975.

Lloyd, J.W., *Foundations of logic programming*, Second, extended edition. Springer-Verlag, 1987.

Marek, W., A natural semantics for modal logic over databases, *Theoretical Computer Science* 56, pp. 187–209, 1988.

Martelli, A. and Montanari, U., An efficient unification algorithm. *ACM Transaction on Programming Languages and Systems* 4, pp. 285–282, 1982.

Mellish, C.S., An alternative to structure sharing in the implementation of a Prolog interpreter. *Logic Programming*, Clark, K.L. and Tärnlund, S.Å., eds., pp. 99–106, Academic Press, 1982.

Miller, D., A theory of modules for logic programming. *Proc. of Symposium on Logic Programming*, IEEE Computer Society Press, pp. 106–114, 1986.

Miller, D., Lexical scoping as universal quantification, *Proc. of Sixth International Conference on Logic Programming*, eds. Levi, G. and Martelli, M., MIT Press, pp. 268–283, 1989.

Monteiro, L. and Porto, A., Contextual Logic Programming, *Proc. of Sixth International Conference on Logic Programming*, eds. Levi, G. and Martelli, M., MIT Press, pp. 284–302, 1989.

Nait Abdallah, M.A., Ions and local definitions in logic programming, *Lecture Notes in Computer Science* 210, pp. 60–72, Springer-Verlag, 1986.

Nait Abdallah, M.A., A logico-algebraic approach to the model theory of knowledge., *Theoretical Computer Science* 66, pp. 205–232, 1989.

Pareschi, R. and Miller, D., Extending definite clause grammars with scoping constructs, *Proc. of Seventh International Conference on Logic Programming*, eds. Warren, D.H.D. and Szeredi, P., MIT Press, pp. 373–389, 1990.

Robinson, A., Forcing in Model Theory, *Proc. Symp. Mat. Instituto Nazionale i Alta Matematica* 5, pp. 64–80, 1971.

Robinson, J.A., A machine-oriented logic based on the resolution principle. *Journal of the ACM* 12, pp. 23–41, 1965.

Shoenfield, J.R., *Mathematical Logic*. Addison-Wesley, 1967.

Smoryński, C.A., Applications of Kripke models, Lecture Notes in Mathematics 344, pp. 324–391, Springer-Verlag, 1973.

Smoryński, C., Modal logic and self-reference, *Handbook of Philosophical Logic*, Gabbay, D. and Guenthner, F., eds., vol. II, pp. 441–495, 1984.

Smoryński, C., *Self-Reference and Modal Logic*. Universitext Series, Springer-Verlag, 1985.

Subrahmanian, V.S., A simple formulation of the theory of metalogic programming. *In:* Abramson, Rogers, 1989, pp. 65–101.

Sugano, H., Meta and reflective computation in logic programs and its semantics. *In:* Bruynooghe, 1990, pp. 19–34.

Warren, D.S., Database updates in Prolog, *Proc. of International Conference on Fifth Generation Systems*, pp. 244–253, 1984.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique.

ISSN 0249 - 6399